LLVM:

Implementing

a

Language

Table of Contents

1. Introduction
2. Lexer
3. Parser
4. Code Generation
5. JIT and Optimizations
6. Control Flow
7. User-Defined Operations
8. Mutable Variables
9. Conclusion

Tutorial Introduction

Welcome to the “Implementing a language with LLVM” tutorial. This tutorial runs through the implementation of a simple language, showing how fun and easy it can be. This tutorial will get you up and started as well as help to build a framework you can extend to other languages. The code in this tutorial can also be used as a playground to hack on other LLVM specific things.

The goal of this tutorial is to progressively unveil our language, describing how it is built up over time. This will let us cover a fairly broad range of language design and LLVM-specific usage issues, showing and explaining the code for it all along the way, without overwhelming you with tons of details up front.

It is useful to point out ahead of time that this tutorial is really about teaching compiler techniques and LLVM specifically, not about teaching modern and sane software engineering principles. In practice, this means that we’ll take a number of shortcuts to simplify the exposition. For example, the code leaks memory, uses global variables all over the place, doesn’t use nice design patterns like visitors, etc... but it is very simple. If you dig in and use the code as a basis for future projects, fixing these deficiencies shouldn’t be hard.

I’ve tried to put this tutorial together in a way that makes chapters easy to skip over if you are already familiar with or are uninterested in the various pieces. The structure of the tutorial is:

- **Chapter #1**: Introduction to the Kaleidoscope language, and the definition of its Lexer - This shows where we are going and the basic functionality that we want it to do. In order to make this tutorial maximally understandable and hackable, we choose to implement everything in C++ instead of using lexer and parser generators. LLVM obviously works just fine with such tools, feel free to use one if you prefer.
- **Chapter #2**: Implementing a Parser and AST - With the lexer in place, we can talk about parsing techniques and basic AST construction. This tutorial describes recursive descent parsing and operator precedence parsing. Nothing in Chapters 1 or 2 is LLVM-specific, the code doesn’t even link in LLVM at this point. :)
- **Chapter #3**: Code generation to LLVM IR - With the AST ready, we can show off how easy generation of LLVM IR really is.
- **Chapter #4**: Adding JIT and Optimizer Support - Because a lot of people are interested in using LLVM as a JIT, we’ll dive right into it and show you the 3 lines it takes to add JIT support. LLVM is also useful in many other ways, but this is one simple and “sexy” way to show off its power. :)
- **Chapter #5**: Extending the Language: Control Flow - With the language up and running, we show how to extend it with control flow operations (if/then/else and a ‘for’ loop). This gives us a chance to talk about simple SSA construction and control flow.
- **Chapter #6**: Extending the Language: User-defined Operators - This is a silly but fun chapter that talks about extending the language to let the user program define their own arbitrary unary and binary operators (with assignable precedence!). This lets us build a significant piece of the “language” as library routines.
- **Chapter #7**: Extending the Language: Mutable Variables - This chapter talks about adding user-defined local variables along with an assignment operator. The interesting part about this is how easy and trivial it is to construct SSA form in LLVM: no, LLVM does not require your front-end to construct SSA form!
- **Chapter #8**: Conclusion and other useful LLVM tidbits - This chapter wraps up the series by talking about potential ways to extend the language, but also includes a bunch of pointers to info about “special topics” like adding garbage collection support, exceptions, debugging, support for “spaghetti stacks”, and a bunch of other tips and tricks.

By the end of the tutorial, we’ll have written a bit less than 700 lines of non-comment, non-blank, lines of code. With this small amount of code, we’ll have built up a very reasonable compiler for a non-trivial language including a hand-written lexer, parser, AST, as well as code generation support with a JIT compiler. While other systems may have interesting “hello world” tutorials, I think the breadth of this tutorial is a great testament to the strengths of LLVM and why you should consider it if you’re interested in language or compiler design.

A note about this tutorial: we expect you to extend the language and play with it on your own. Take the code and go crazy hacking away at it, compilers don’t need to be scary creatures - it can be a lot of fun to play with languages!

Kaleidoscope Language and its Lexer

1.1 The Basic Language

This tutorial will be illustrated with a toy language that we'll call "Kaleidoscope" (derived from "meaning beautiful, form, and view"). Kaleidoscope is a procedural language that allows you to define functions, use conditionals, math, etc. Over the course of the tutorial, we'll extend Kaleidoscope to support the if/then/else construct, a for loop, user defined operators, JIT compilation with a simple command line interface, etc.

Because we want to keep things simple, the only datatype in Kaleidoscope is a 64-bit floating point type (aka `double` in C parlance). As such, all values are implicitly double precision and the language doesn't require type declarations. This gives the language a very nice and simple syntax. For example, the following simple example computes Fibonacci numbers:

```
# Compute the x'th fibonacci number.
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
```

We also allow Kaleidoscope to call into standard library functions (the LLVM JIT makes this completely trivial). This means that you can use the `extern` keyword to define a function before you use it (this is also useful for mutually recursive functions). For example:

```
extern sin(arg);
extern cos(arg);
extern atan2(arg1 arg2);

atan2(sin(.4), cos(42))
```

A more interesting example is included in Chapter 6 where we write a little Kaleidoscope application that displays a Mandelbrot Set at various levels of magnification.

Lets dive into the implementation of this language!

1.2 The Lexer

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a "lexer" (aka 'scanner') to break the input up into "tokens". Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities:

```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number
```

Each token returned by our lexer will either be one of the Token enum values or it will be an 'unknown' character like `+`, which is returned as its ASCII value. If the current token is an identifier, the IdentifierStr global variable holds the name of the identifier. If the current token is a numeric literal (like 1.0), NumVal holds its value. Note that we use global variables for simplicity, this is not the best choice for a real language implementation :).

The actual implementation of the lexer is a single function named gettok. The gettok function is called to return the next token from standard input. Its definition starts as:

```
/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();
```

gettok works by calling the C getchar() function to read characters one at a time from standard input. It eats them as it recognizes them and stores the last character read, but not processed, in LastChar. The first thing that it has to do is ignore whitespace between tokens. This is accomplished with the loop above.

The next thing gettok needs to do is recognize identifiers and specific keywords like `def`. Kaleidoscope does this with this simple loop:

```
if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
  IdentifierStr = LastChar;
  while (isalnum((LastChar = getchar())))
    IdentifierStr += LastChar;

  if (IdentifierStr == "def") return tok_def;
  if (IdentifierStr == "extern") return tok_extern;
  return tok_identifier;
}
```

Note that this code sets the `IdentifierStr` global whenever it lexes an identifier. Also, since language keywords are matched by the same loop, we handle them here inline. Numeric values are similar:

```
if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}
```

This is all pretty straight-forward code for processing input. When reading a numeric value from input, we use the C strtod

function to convert it to a numeric value that we store in NumVal. Note that this isn't doing sufficient error checking: it will incorrectly read "1.23.45.67" and handle it as if you typed in "1.23". Feel free to extend it :). Next we handle comments:

```
if (LastChar == '#') {
  // Comment until end of line.
  do LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn't match one of the above cases, it is either an operator character like `+` or the end of the file. These are handled with this code:

```
  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}
```

With this, we have the complete lexer for the basic Kaleidoscope language (the full code listing for the Lexer is available in the next chapter of the tutorial). Next we'll build a simple parser that uses this to build an Abstract Syntax Tree. When we have that, we'll include a driver so that you can use the lexer and parser together.

Implementing a Parser and AST

Welcome to Chapter 2 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to use the lexer, built in Chapter 1, to build a full parser for our Kaleidoscope language. Once we have a parser, we'll define and build an Abstract Syntax Tree (AST).

The parser we will build uses a combination of Recursive Descent Parsing and Operator-Precedence Parsing to parse the Kaleidoscope language (the latter for binary expressions and the former for everything else). Before we get to parsing though, lets talk about the output of the parser: the Abstract Syntax Tree.

2.1 The Abstract Syntax Tree (AST)

The AST for a program captures its behavior in such a way that it is easy for later stages of the compiler (e.g. code generation) to interpret. We basically want one object for each construct in the language, and the AST should closely model the language. In Kaleidoscope, we have expressions, a prototype, and a function object. We'll start with expressions first:

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
};
```

The code above shows the definition of the base ExprAST class and one subclass which we use for numeric literals. The important thing to note about this code is that the NumberExprAST class captures the numeric value of the literal as an instance variable. This allows later phases of the compiler to know what the stored numeric value is.

Right now we only create the AST, so there are no useful accessor methods on them. It would be very easy to add a virtual method to pretty print the code, for example. Here are the other expression AST node definitions that we'll use in the basic form of the Kaleidoscope language:

```
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
};
```

This is all (intentionally) rather straight-forward: variables capture the variable name, binary operators capture their opcode (e.g. `+`), and calls capture a function name as well as a list of any argument expressions. One thing that is nice about our AST is that it captures the language features without talking about the syntax of the language. Note that there is no discussion about precedence of binary operators, lexical structure, etc.

For our basic language, these are all of the expression nodes we'll define. Because it doesn't have conditional control flow, it isn't Turing-complete; we'll fix that in a later installment. The two things we need next are a way to talk about the interface to a function, and a way to talk about functions themselves:

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args)
    : Name(name), Args(args) {}
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}
};
```

In Kaleidoscope, functions are typed with just a count of their arguments. Since all values are double precision floating point, the type of each argument doesn't need to be stored anywhere. In a more aggressive and realistic language, the `ExprAST` class would probably have a type field.

With this scaffolding, we can now talk about parsing expressions and function bodies in Kaleidoscope.

2.2 Parser Basics

Now that we have an AST to build, we need to define the parser code to build it. The idea here is that we want to parse something like `x+y` (which is returned as three tokens by the lexer) into an AST that could be generated with calls like this:

```
ExprAST *X = new VariableExprAST("x");
ExprAST *Y = new VariableExprAST("y");
ExprAST *Result = new BinaryExprAST('+', X, Y);
In order to do this, we'll start by defining some basic helper routines:

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}
```

This implements a simple token buffer around the lexer. This allows us to look one token ahead at what the lexer is returning. Every function in our parser will assume that CurTok is the current token that needs to be parsed.

```
/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }
```

The Error routines are simple helper routines that our parser will use to handle errors. The error recovery in our parser will not be the best and is not particular user-friendly, but it will be enough for our tutorial. These routines make it easier to handle errors in routines that have various return types: they always return null.

With these basic helper functions, we can implement the first piece of our grammar: numeric literals.

2.3 Basic Expression Parsing

We start with numeric literals, because they are the simplest to process. For each production in our grammar, we'll define a function which parses that production. For numeric literals, we have:

```
/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}
```

This routine is very simple: it expects to be called when the current token is a tok_number token. It takes the current number value, creates a NumberExprAST node, advances the lexer to the next token, and finally returns.

There are some interesting aspects to this. The most important one is that this routine eats all of the tokens that correspond to the production and returns the lexer buffer with the next token (which is not part of the grammar production) ready to go. This is a fairly standard way to go for recursive descent parsers. For a better example, the parenthesis operator is defined like this:

```
/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}
```

This function illustrates a number of interesting things about the parser:

1) It shows how we use the Error routines. When called, this function expects that the current token is a `(` token, but after parsing the subexpression, it is possible that there is no `)` waiting. For example, if the user types in `(4 x` instead of `(4)`, the parser should emit an error. Because errors can occur, the parser needs a way to indicate that they happened: in our parser, we return null on an error.

2) Another interesting aspect of this function is that it uses recursion by calling ParseExpression (we will soon see that ParseExpression can call ParseParenExpr). This is powerful because it allows us to handle recursive grammars, and keeps each production very simple. Note that parentheses do not cause construction of AST nodes themselves. While we could do it this way, the most important role of parentheses are to guide the parser and provide grouping. Once the parser constructs the AST, parentheses are not needed.

The next simple production is for handling variable references and function calls:

```
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}
```

This routine follows the same style as the other routines. (It expects to be called if the current token is a tok_identifier token). It also has recursion and error handling. One interesting aspect of this is that it uses look-ahead to determine if the current identifier is a stand alone variable reference or if it is a function call expression. It handles this by checking to see if the token after the identifier is a `(` token, constructing either a VariableExprAST or CallExprAST node as appropriate.

Now that we have all of our simple expression-parsing logic in place, we can define a helper function to wrap it together into one entry point. We call this class of expressions "primary" expressions, for reasons that will become more clear later in the tutorial. In order to parse an arbitrary primary expression, we need to determine what sort of expression it is:

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  }
}
```

Now that you see the definition of this function, it is more obvious why we can assume the state of CurTok in the various functions. This uses look-ahead to determine which sort of expression is being inspected, and then parses it with a function call.

Now that basic expressions are handled, we need to handle binary expressions. They are a bit more complex.

2.4 Binary Expression Parsing

Binary expressions are significantly harder to parse because they are often ambiguous. For example, when given the string `x+y*z`, the parser can choose to parse it as either `(x+y)*z` or `x+(y*z)`. With common definitions from mathematics, we expect the later parse, because `*` (multiplication) has higher precedence than `+` (addition).

There are many ways to handle this, but an elegant and efficient way is to use Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion. To start with, we need a table of precedences:

```
/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.
  ...
}
```

For the basic form of Kaleidoscope, we will only support 4 binary operators (this can obviously be extended by you, our brave and intrepid reader). The GetTokPrecedence function returns the precedence for the current token, or -1 if the token is not a binary operator. Having a map makes it easy to add new operators and makes it clear that the algorithm doesn't depend on the specific operators involved, but it would be easy enough to eliminate the map and do the comparisons in the GetTokPrecedence function. (Or just use a fixed-size array).

With the helper above defined, we can now start parsing binary expressions. The basic idea of operator precedence parsing is to break down an expression with potentially ambiguous binary operators into pieces. Consider ,for example, the expression `a+b+(c+d)*e*f+g`. Operator precedence parsing considers this as a stream of primary expressions separated by binary operators. As such, it will first parse the leading primary expression `a`, then it will see the pairs [+, b] [+, (c+d)] [*, e] [*, f] and [+, g]. Note that because parentheses are primary expressions, the binary expression parser doesn't need to worry about nested subexpressions like (c+d) at all.

To start, an expression is a primary expression potentially followed by a sequence of [binop,primaryexpr] pairs:

```
/// expression
///   ::= primary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParsePrimary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}
```

ParseBinOpRHS is the function that parses the sequence of pairs for us. It takes a precedence and a pointer to an expression for the part that has been parsed so far. Note that `x` is a perfectly valid expression: As such, `binoprhs` is allowed to be empty, in which case it returns the expression that is passed into it. In our example above, the code passes the expression for `a` into ParseBinOpRHS and the current token is `+`.

The precedence value passed into ParseBinOpRHS indicates the minimal operator precedence that the function is allowed to eat. For example, if the current pair stream is [+, x] and ParseBinOpRHS is passed in a precedence of 40, it will not consume any tokens (because the precedence of `+` is only 20). With this in mind, ParseBinOpRHS starts with:

```
/// binoprhs
///   ::= ('+' primary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;
```

This code gets the precedence of the current token and checks to see if if is too low. Because we defined invalid tokens to have a precedence of -1, this check implicitly knows that the pair-stream ends when the token stream runs out of binary operators. If this check succeeds, we know that the token is a binary operator and that it will be included in this expression:

```
// Okay, we know this is a binop.
int BinOp = CurTok;
getNextToken();  // eat binop

// Parse the primary expression after the binary operator.
ExprAST *RHS = ParsePrimary();
if (!RHS) return 0;
```

As such, this code eats (and remembers) the binary operator and then parses the primary expression that follows. This builds up the whole pair, the first of which is [+, b] for the running example.

Now that we parsed the left-hand side of an expression and one pair of the RHS sequence, we have to decide which way the expression associates. In particular, we could have `(a+b) binop unparsed` or `a + (b binop unparsed)`. To determine this, we look ahead at `binop` to determine its precedence and compare it to BinOp's precedence (which is `+` in this case):

```
// If BinOp binds less tightly with RHS than the operator after RHS, let
// the pending operator take RHS as its LHS.
int NextPrec = GetTokPrecedence();
if (TokPrec < NextPrec) {
```

If the precedence of the binop to the right of "RHS" is lower or equal to the precedence of our current operator, then we know that the parentheses associate as `(a+b) binop ...`. In our example, the current operator is `+` and the next operator is `+`, we know that they have the same precedence. In this case we'll create the AST node for `a+b`, and then continue parsing:

```
      ... if body omitted ...
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }  // loop around to the top of the while loop.
}
```

In our example above, this will turn `a+b+` into `(a+b)` and execute the next iteration of the loop, with `+` as the current token. The code above will eat, remember, and parse `(c+d)` as the primary expression, which makes the current pair equal to [+, (c+d)]. It will then evaluate the `if` conditional above with `*` as the binop to the right of the primary. In this case, the precedence of `*` is higher than the precedence of `+` so the if condition will be entered.

The critical question left here is "how can the if condition parse the right hand side in full"? In particular, to build the AST correctly for our example, it needs to get all of `(c+d)*e*f` as the RHS expression variable. The code to do this is surprisingly simple (code from the above two blocks duplicated for context):

```
    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }
    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }  // loop around to the top of the while loop.
}
```

At this point, we know that the binary operator to the RHS of our primary has higher precedence than the binop we are currently parsing. As such, we know that any sequence of pairs whose operators are all higher precedence than `+` should be parsed together and returned as "RHS". To do this, we recursively invoke the ParseBinOpRHS function specifying "TokPrec+1" as the minimum precedence required for it to continue. In our example above, this will cause it to return the AST node for `(c+d)*e*f` as RHS, which is then set as the RHS of the `+` expression.

Finally, on the next iteration of the while loop, the `+g` piece is parsed and added to the AST. With this little bit of code (14 non-trivial lines), we correctly handle fully general binary expression parsing in a very elegant way. This was a whirlwind tour of this code, and it is somewhat subtle. I recommend running through it with a few tough examples to see how it works.

This wraps up handling of expressions. At this point, we can point the parser at an arbitrary token stream and build an expression from it, stopping at the first token that is not part of the expression. Next up we need to handle function definitions, etc.

2.5 Parsing the Rest

The next thing missing is handling of function prototypes. In Kaleidoscope, these are used both for `extern` function declarations as well as function body definitions. The code to do this is straight-forward and not very interesting (once you've survived expressions):

```
/// prototype
///   ::= id '(' id* ')'
static PrototypeAST *ParsePrototype() {
  if (CurTok != tok_identifier)
    return ErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  // Read the list of argument names.
  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  return new PrototypeAST(FnName, ArgNames);
}
```

Given this, a function definition is very simple, just a prototype plus an expression to implement the body:

```
/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}
```

In addition, we support `extern` to declare functions like `sin` and `cos` as well as to support forward declaration of user functions. These ‘extern's are just prototypes with no body:

```
/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}
```

Finally, we'll also let the user type in arbitrary top-level expressions and evaluate them on the fly. We will handle this by defining anonymous nullary (zero argument) functions for them:

```
/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}
```

Now that we have all the pieces, let's build a little driver that will let us actually execute this code we've built!

2.6 The Driver

The driver for this simply invokes all of the parsing pieces with a top-level dispatch loop. There isn't much interesting here, so I'll just include the top-level loop. See below for full code in the "Top-Level Parsing" section.

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}
```

The most interesting part of this is that we ignore top-level semicolons. Why is this, you ask? The basic reason is that if you type `4 + 5` at the command line, the parser doesn't know whether that is the end of what you will type or not. For example, on the next line you could type `def foo...` in which case `4+5` is the end of a top-level expression. Alternatively you could type `* 6`, which would continue the expression. Having top-level semicolons allows you to type `4+5;`, and the parser will know you are done.

2.8. Conclusions

With just under 400 lines of commented code (240 lines of non-comment, non-blank code), we fully defined our minimal language, including a lexer, parser, and AST builder. With this done, the executable will validate Kaleidoscope code and tell us if it is grammatically invalid. For example, here is a sample interaction:

```
$ ./a.out
ready> def foo(x y) x+foo(y, 4.0);
Parsed a function definition.
ready> def foo(x y) x+y y;
Parsed a function definition.
Parsed a top-level expr
ready> def foo(x y) x+y );
Parsed a function definition.
Error: unknown token when expecting an expression
ready> extern sin(a);
ready> Parsed an extern
ready> ^D
$
```

There is a lot of room for extension here. You can define new AST nodes, extend the language in many ways, etc. In the next installment, we will describe how to generate LLVM Intermediate Representation (IR) from the AST.

Code Generation to LLVM IR

Welcome to Chapter 3 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to transform the Abstract Syntax Tree, built in Chapter 2, into LLVM IR. This will teach you a little bit about how LLVM does things, as well as demonstrate how easy it is to use. It's much more work to build a lexer and parser than it is to generate LLVM IR code. :)

Please note: the code in this chapter and later require LLVM 2.2 or later. LLVM 2.1 and before will not work with it. Also note that you need to use a version of this tutorial that matches your LLVM release: If you are using an official LLVM release, use the version of the documentation included with your release or on the llvm.org releases page.

3.1 Code Generation Setup

In order to generate LLVM IR, we want some simple setup to get started. First we define virtual code generation (codegen) methods in each AST class:

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};
...
```

The `Codegen()` method says to emit IR for that AST node along with all the things it depends on, and they all return an LLVM Value object. "Value" is the class used to represent a "Static Single Assignment (SSA) register" or "SSA value" in LLVM. The most distinct aspect of SSA values is that their value is computed as the related instruction executes, and it does not get a new value until (and if) the instruction re-executes. In other words, there is no way to "change" an SSA value. For more information, please read up on Static Single Assignment - the concepts are really quite natural once you grok them.

Note that instead of adding virtual methods to the ExprAST class hierarchy, it could also make sense to use a visitor pattern or some other way to model this. Again, this tutorial won't dwell on good software engineering practices: for our purposes, adding a virtual method is simplest.

The second thing we want is an `Error` method like we used for the parser, which will be used to report errors found during code generation (for example, use of an undeclared parameter):

```
Value *ErrorV(const char *Str) { Error(Str); return 0; }

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
```

The static variables will be used during code generation. TheModule is the LLVM construct that contains all of the functions and global variables in a chunk of code. In many ways, it is the top-level structure that the LLVM IR uses to contain code.

The Builder object is a helper object that makes it easy to generate LLVM instructions. Instances of the IRBuilder class template keep track of the current place to insert instructions and has methods to create new instructions.

The NamedValues map keeps track of which values are defined in the current scope and what their LLVM representation is. (In other words, it is a symbol table for the code). In this form of Kaleidoscope, the only things that can be referenced are function parameters. As such, function parameters will be in this map when generating code for their function body.

With these basics in place, we can start talking about how to generate code for each expression. Note that this assumes that the Builder has been set up to generate code into something. For now, we'll assume that this has already been done, and we'll just use it to emit code.

3.2 Expression Code Generation

Generating LLVM code for expression nodes is very straightforward: less than 45 lines of commented code for all four of our expression nodes. First we'll do numeric literals:

```
Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}
```

In the LLVM IR, numeric constants are represented with the ConstantFP class, which holds the numeric value in an APFloat internally (APFloat has the capability of holding floating point constants of Arbitrary Precision). This code basically just creates and returns a ConstantFP. Note that in the LLVM IR that constants are all uniqued together and shared. For this reason, the API uses the `foo::get(...)` idiom instead of `new foo(..)` or `foo::Create(..)`.

```
Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  return V ? V : ErrorV("Unknown variable name");
}
```

References to variables are also quite simple using LLVM. In the simple version of Kaleidoscope, we assume that the variable has already been emitted somewhere and its value is available. In practice, the only values that can be in the NamedValues map are function arguments. This code simply checks to see that the specified name is in the map (if not, an unknown variable is being referenced) and returns the value for it. In future chapters, we'll add support for loop induction variables in the symbol table, and for local variables.

```
Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: return ErrorV("invalid binary operator");
  }
}
```

Binary operators start to get more interesting. The basic idea here is that we recursively emit code for the left-hand side of the expression, then the right-hand side, then we compute the result of the binary expression. In this code, we do a simple switch on the opcode to create the right LLVM instruction.

In the example above, the LLVM builder class is starting to show its value. IRBuilder knows where to insert the newly created instruction, all you have to do is specify what instruction to create (e.g. with CreateFAdd), which operands to use (L and R here) and optionally provide a name for the generated instruction.

One nice thing about LLVM is that the name is just a hint. For instance, if the code above emits multiple `addtmp` variables, LLVM will automatically provide each one with an increasing, unique numeric suffix. Local value names for instructions are purely optional, but it makes it much easier to read the IR dumps.

LLVM instructions are constrained by strict rules: for example, the Left and Right operators of an add instruction must have the same type, and the result type of the add must match the operand types. Because all values in Kaleidoscope are doubles, this makes for very simple code for add, sub and mul.

On the other hand, LLVM specifies that the fcmp instruction always returns an 'i1' value (a one bit integer). The problem with this is that Kaleidoscope wants the value to be a 0.0 or 1.0 value. In order to get these semantics, we combine the fcmp instruction with a uitofp instruction. This instruction converts its input integer into a floating point value by treating the input as an unsigned value. In contrast, if we used the sitofp instruction, the Kaleidoscope '<' operator would return 0.0 and -1.0, depending on the input value.

```
Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");

  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

Code generation for function calls is quite straightforward with LLVM. The code above initially does a function name lookup in the LLVM Module's symbol table. Recall that the LLVM Module is the container that holds all of the functions we are JIT'ing. By giving each function the same name as what the user specifies, we can use the LLVM symbol table to resolve function names for us.

Once we have the function to call, we recursively codegen each argument that is to be passed in, and create an LLVM call instruction. Note that LLVM uses the native C calling conventions by default, allowing these calls to also call into standard library functions like `sin` and `cos`, with no additional effort.

This wraps up our handling of the four basic expressions that we have so far in Kaleidoscope. Feel free to go in and add some more. For example, by browsing the LLVM language reference you'll find several other interesting instructions that are really easy to plug into our basic framework.

3.3 Function Code Generation

Code generation for prototypes and functions must handle a number of details, which make their code less beautiful than expression code generation, but allows us to illustrate some important points. First, lets talk about code generation for prototypes: they are used both for function bodies and external function declarations. The code starts with:

```
Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);
```

This code packs a lot of power into a few lines. Note first that this function returns a `Function*` instead of a `Value*`. Because a "prototype" really talks about the external interface for a function (not the value computed by an expression), it makes sense for it to return the LLVM Function it corresponds to when codegen'd.

The call to FunctionType::get creates the FunctionType that should be used for a given Prototype. Since all function arguments in Kaleidoscope are of type double, the first line creates a vector of "N" LLVM double types. It then uses the Functiontype::get method to create a function type that takes "N" doubles as arguments, returns one double as a result, and that is not vararg (the false parameter indicates this). Note that Types in LLVM are uniqued just like Constants are, so you don't "new" a type, you "get" it.

The final line above actually creates the function that the prototype will correspond to. This indicates the type, linkage and name to use, as well as which module to insert into. "external linkage" means that the function may be defined outside the current module and/or that it is callable by functions outside the module. The Name passed in is the name the user specified: since "TheModule" is specified, this name is registered in "TheModule"s symbol table, which is used by the function call code above.

```
// If F conflicted, there was already something named 'Name'.  If it has a
// body, don't allow redefinition or reextern.
if (F->getName() != Name) {
  // Delete the one we just made and get the existing one.
  F->eraseFromParent();
  F = TheModule->getFunction(Name);
```

The Module symbol table works just like the Function symbol table when it comes to name conflicts: if a new function is created with a name that was previously added to the symbol table, the new function will get implicitly renamed when added to the Module. The code above exploits this fact to determine if there was a previous definition of this function.

In Kaleidoscope, I choose to allow redefinitions of functions in two cases: first, we want to allow `extern`ing a function more than once, as long as the prototypes for the externs match (since all arguments have the same type, we just have to check that the number of arguments match). Second, we want to allow `extern`ing a function and then defining a body for it. This is useful when defining mutually recursive functions.

In order to implement this, the code above first checks to see if there is a collision on the name of the function. If so, it deletes the function we just created (by calling eraseFromParent) and then calling getFunction to get the existing function with the specified name. Note that many APIs in LLVM have "erase" forms and "remove" forms. The "remove" form unlinks the object from its parent (e.g. a Function from a Module) and returns it. The "erase" form unlinks the object and then deletes it.

```
  // If F already has a body, reject this.
  if (!F->empty()) {
    ErrorF("redefinition of function");
    return 0;
  }

  // If F took a different number of args, reject.
  if (F->arg_size() != Args.size()) {
    ErrorF("redefinition of function with different # args");
    return 0;
  }
}
```

In order to verify the logic above, we first check to see if the pre-existing function is "empty". In this case, empty means that it has no basic blocks in it, which means it has no body. If it has no body, it is a forward declaration. Since we don't allow anything after a full definition of the function, the code rejects this case. If the previous reference to a function was an ‘extern', we simply verify that the number of arguments for that definition and this one match up. If not, we emit an error.

```
  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx) {
    AI->setName(Args[Idx]);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = AI;
  }
  return F;
}
```

The last bit of code for prototypes loops over all of the arguments in the function, setting the name of the LLVM Argument objects to match, and registering the arguments in the NamedValues map for future use by the VariableExprAST AST node. Once this is set up, it returns the Function object to the caller. Note that we don't check for conflicting argument names here (e.g. "extern foo(a b a)"). Doing so would be very straight-forward with the mechanics we have already used above.

```
Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;
```

Code generation for function definitions starts out simply enough: we just codegen the prototype (Proto) and verify that it is ok. We then clear out the NamedValues map to make sure that there isn't anything in it from the last function we compiled. Code generation of the prototype ensures that there is an LLVM Function object that is ready to go for us.

```
// Create a new basic block to start insertion into.
BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
Builder.SetInsertPoint(BB);

if (Value *RetVal = Body->Codegen()) {
```

Now we get to the point where the Builder is set up. The first line creates a new basic block (named "entry"), which is inserted into TheFunction. The second line then tells the builder that new instructions should be inserted into the end of the new basic block. Basic blocks in LLVM are an important part of functions that define the Control Flow Graph. Since we don't have any control flow, our functions will only contain one block at this point. We'll fix this in Chapter 5 :).

```
if (Value *RetVal = Body->Codegen()) {
  // Finish off the function.
  Builder.CreateRet(RetVal);

  // Validate the generated code, checking for consistency.
  verifyFunction(*TheFunction);

  return TheFunction;
}
```

Once the insertion point is set up, we call the CodeGen() method for the root expression of the function. If no error happens, this emits code to compute the expression into the entry block and returns the value that was computed. Assuming no error, we then create an LLVM ret instruction, which completes the function. Once the function is built, we call verifyFunction, which is provided by LLVM. This function does a variety of consistency checks on the generated code, to determine if our compiler is doing everything right. Using this is important: it can catch a lot of bugs. Once the function is finished and validated, we return it.

```
    // Error reading body, remove function.
    TheFunction->eraseFromParent();
    return 0;
  }
```

The only piece left here is handling of the error case. For simplicity, we handle this by merely deleting the function we produced with the eraseFromParent method. This allows the user to redefine a function that they incorrectly typed in before: if we didn't delete it, it would live in the symbol table, with a body, preventing future redefinition.

This code does have a bug, though. Since the `PrototypeAST::Codegen` can return a previously defined forward declaration, our code can actually delete a forward declaration. There are a number of ways to fix this bug, see what you can come up with! Here is a testcase:

```
extern foo(a b);     # ok, defines foo.
def foo(a b) c;      # error, 'c' is invalid.
def bar() foo(1, 2); # error, unknown function "foo"
```

3.4 Driver Changes and Closing Thoughts

For now, code generation to LLVM doesn't really get us much, except that we can look at the pretty IR calls. The sample code inserts calls to Codegen into the `HandleDefinition`, `HandleExtern` etc functions, and then dumps out the LLVM IR. This gives a nice way to look at the LLVM IR for simple functions. For example:

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}
```

Note how the parser turns the top-level expression into anonymous functions for us. This will be handy when we add JIT support in the next chapter. Also note that the code is very literally transcribed, no optimizations are being performed except simple constant folding done by IRBuilder. We will add optimizations explicitly in the next chapter.

```
ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}
```

This shows some simple arithmetic. Notice the striking similarity to the LLVM builder calls that we use to create the instructions.

```
ready> def bar(a) foo(a, 4.0) + bar(31337);
Read function definition:
define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}
```

This shows some function calls. Note that this function will take a long time to execute if you call it. In the future we'll add conditional control flow to actually make recursion useful :).

```
ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> cos(1.234);
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

This shows an extern for the libm "cos" function, and a call to it.

```
ready> ^D
; ModuleID = 'my cool jit'

define double @0() {
entry:
  %addtmp = fadd double 4.000000e+00, 5.000000e+00
  ret double %addtmp
}

define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}

define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}

declare double @cos(double)

define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

When you quit the current demo, it dumps out the IR for the entire module generated. Here you can see the big picture with all the functions referencing each other.

This wraps up the third chapter of the Kaleidoscope tutorial. Up next, we'll describe how to add JIT codegen and optimizer support to this so we can actually start running code!

Adding JIT and Optimizer Support

Welcome to Chapter 4 of the "Implementing a language with LLVM" tutorial. Chapters 1-3 described the implementation of a simple language and added support for generating LLVM IR. This chapter describes two new techniques: adding optimizer support to your language, and adding JIT compiler support. These additions will demonstrate how to get nice, efficient code for the Kaleidoscope language.

4.1 Trivial Constant Folding

Our demonstration for Chapter 3 is elegant and easy to extend. Unfortunately, it does not produce wonderful code. The IRBuilder, however, does give us obvious optimizations when compiling simple code:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        ret double %addtmp
}
```

This code is not a literal transcription of the AST built by parsing the input. That would be:

```
ready> def test(x) 1+2+x;
Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 2.000000e+00, 1.000000e+00
        %addtmp1 = fadd double %addtmp, %x
        ret double %addtmp1
}
```

Constant folding, as seen above, in particular, is a very common and very important optimization: so much so that many language implementors implement constant folding support in their AST representation.

With LLVM, you don't need this support in the AST. Since all calls to build LLVM IR go through the LLVM IR builder, the builder itself checked to see if there was a constant folding opportunity when you call it. If so, it just does the constant fold and return the constant instead of creating an instruction.

Well, that was easy :). In practice, we recommend always using IRBuilder when generating code like this. It has no "syntactic overhead" for its use (you don't have to uglify your compiler with constant checks everywhere) and it can dramatically reduce the amount of LLVM IR that is generated in some cases (particular for languages with a macro preprocessor or that use a lot of constants).

On the other hand, the IRBuilder is limited by the fact that it does all of its analysis inline with the code as it is built. If you take a slightly more complex example:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        %addtmp1 = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp1
        ret double %multmp
}
```

In this case, the LHS and RHS of the multiplication are the same value. We'd really like to see this generate `tmp = x+3; result = tmp*tmp;` instead of computing `x+3` twice.

Unfortunately, no amount of local analysis will be able to detect and correct this. This requires two transformations: reassociation of expressions (to make the add's lexically identical) and Common Subexpression Elimination (CSE) to delete the redundant add instruction. Fortunately, LLVM provides a broad range of optimizations that you can use, in the form of "passes".

4.2 LLVM Optimization Passes

LLVM provides many optimization passes, which do many different sorts of things and have different tradeoffs. Unlike other systems, LLVM doesn't hold to the mistaken notion that one set of optimizations is right for all languages and for all situations. LLVM allows a compiler implementor to make complete decisions about what optimizations to use, in which order, and in what situation.

As a concrete example, LLVM supports both "whole module" passes, which look across as large of body of code as they can (often a whole file, but if run at link time, this can be a substantial portion of the whole program). It also supports and includes "per-function" passes which just operate on a single function at a time, without looking at other functions. For more information on passes and how they are run, see the How to Write a Pass document and the List of LLVM Passes.

For Kaleidoscope, we are currently generating functions on the fly, one at a time, as the user types them in. We aren't shooting for the ultimate optimization experience in this setting, but we also want to catch the easy and quick stuff where possible. As such, we will choose to run a few per-function optimizations as the user types the function in. If we wanted to make a "static Kaleidoscope compiler", we would use exactly the code we have now, except that we would defer running the optimizer until the entire file has been parsed.

In order to get per-function optimizations going, we need to set up a FunctionPassManager to hold and organize the LLVM optimizations that we want to run. Once we have that, we can add a set of optimizations to run. The code looks like this:

```
FunctionPassManager OurFPM(TheModule);

// Set up the optimizer pipeline.  Start with registering info about how the
// target lays out data structures.
OurFPM.add(new DataLayout(*TheExecutionEngine->getDataLayout()));
// Provide basic AliasAnalysis support for GVN.
OurFPM.add(createBasicAliasAnalysisPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
// Reassociate expressions.
OurFPM.add(createReassociatePass());
// Eliminate Common SubExpressions.
OurFPM.add(createGVNPass());
// Simplify the control flow graph (deleting unreachable blocks, etc).
OurFPM.add(createCFGSimplificationPass());

OurFPM.doInitialization();

// Set the global so the code gen can use this.
TheFPM = &OurFPM;

// Run the main "interpreter loop" now.
MainLoop();
```

This code defines a FunctionPassManager, `OurFPM`. It requires a pointer to the Module to construct itself. Once it is set up, we use a series of `add` calls to add a bunch of LLVM passes. The first pass is basically boilerplate, it adds a pass so that later optimizations know how the data structures in the program are laid out. The `TheExecutionEngine` variable is related to the JIT, which we will get to in the next section.

In this case, we choose to add 4 optimization passes. The passes we chose here are a pretty standard set of "cleanup" optimizations that are useful for a wide variety of code. I won't delve into what they do but, believe me, they are a good starting place :).

Once the PassManager is set up, we need to make use of it. We do this by running it after our newly created function is constructed (in `FunctionAST::Codegen`), but before it is returned to the client:

```
if (Value *RetVal = Body->Codegen()) {
  // Finish off the function.
  Builder.CreateRet(RetVal);

  // Validate the generated code, checking for consistency.
  verifyFunction(*TheFunction);

  // Optimize the function.
  TheFPM->run(*TheFunction);

  return TheFunction;
}
```

As you can see, this is pretty straightforward. The FunctionPassManager optimizes and updates the LLVM `Function*` in place, improving (hopefully) its body. With this in place, we can try our test above again:

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp
        ret double %multmp
}
```

As expected, we now get our nicely optimized code, saving a floating point add instruction from every execution of this function.

LLVM provides a wide variety of optimizations that can be used in certain circumstances. Some documentation about the various passes is available, but it isn't very complete. Another good source of ideas can come from looking at the passes that Clang runs to get started. The "opt" tool allows you to experiment with passes from the command line, so you can see if they do anything.

Now that we have reasonable code coming out of our front-end, lets talk about executing it!

4.3 Adding a JIT Compiler

Code that is available in LLVM IR can have a wide variety of tools applied to it. For example, you can run optimizations on it (as we did above), you can dump it out in textual or binary forms, you can compile the code to an assembly file (.s) for some target, or you can JIT compile it. The nice thing about the LLVM IR representation is that it is the "common currency" between many different parts of the compiler.

In this section, we'll add JIT compiler support to our interpreter. The basic idea that we want for Kaleidoscope is to have the user enter function bodies as they do now, but immediately evaluate the top-level expressions they type in. For example, if they type in `1 + 2;`, we should evaluate and print out 3. If they define a function, they should be able to call it from the command line.

In order to do this, we first declare and initialize the JIT. This is done by adding a global variable and a call in main:

```
static ExecutionEngine *TheExecutionEngine;
...
int main() {
  ..
  // Create the JIT.  This takes ownership of the module.
  TheExecutionEngine = EngineBuilder(TheModule).create();
  ..
}
```

This creates an abstract `Execution Engine` which can be either a JIT compiler or the LLVM interpreter. LLVM will automatically pick a JIT compiler for you if one is available for your platform, otherwise it will fall back to the interpreter.

Once the ExecutionEngine is created, the JIT is ready to be used. There are a variety of APIs that are useful, but the simplest one is the `getPointerToFunction(F)` method. This method JIT compiles the specified LLVM Function and returns a function pointer to the generated machine code. In our case, this means that we can change the code that parses a top-level expression to look like this:

```
static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      LF->dump();  // Dump the function for exposition purposes.

      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
```

Recall that we compile top-level expressions into a self-contained LLVM function that takes no arguments and returns the computed double. Because the LLVM JIT compiler matches the native platform ABI, this means that you can just cast the result pointer to a function pointer of that type and call it directly. This means, there is no difference between JIT compiled code and native machine code that is statically linked into your application.

With just these two changes, lets see how Kaleidoscope works now!

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}

Evaluated to 9.000000
```

Well this looks like it is basically working. The dump of the function shows the "no argument function that always returns double" that we synthesize for each top-level expression that is typed in. This demonstrates very basic functionality, but can we do more?

```
ready> def testfunc(x y) x + y*2;
Read function definition:
define double @testfunc(double %x, double %y) {
entry:
  %multmp = fmul double %y, 2.000000e+00
  %addtmp = fadd double %multmp, %x
  ret double %addtmp
}

ready> testfunc(4, 10);
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @testfunc(double 4.000000e+00, double 1.000000e+01)
  ret double %calltmp
}

Evaluated to 24.000000
```

This illustrates that we can now call user code, but there is something a bit subtle going on here. Note that we only invoke the JIT on the anonymous functions that call testfunc, but we never invoked it on testfunc itself. What actually happened here is that the JIT scanned for all non-JIT'd functions transitively called from the anonymous function and compiled all of them before returning from `getPointerToFunction()`.

The JIT provides a number of other more advanced interfaces for things like freeing allocated machine code, rejit'ing

functions to update them, etc. However, even with this simple code, we get some surprisingly powerful capabilities - check this out (I removed the dump of the anonymous functions, you should get the idea by now :) :

```
ready> extern sin(x);
Read extern:
declare double @sin(double)

ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> sin(1.0);
Read top-level expression:
define double @2() {
entry:
  ret double 0x3FEAED548F090CEE
}

Evaluated to 0.841471

ready> def foo(x) sin(x)*sin(x) + cos(x)*cos(x);
Read function definition:
define double @foo(double %x) {
entry:
  %calltmp = call double @sin(double %x)
  %multmp = fmul double %calltmp, %calltmp
  %calltmp2 = call double @cos(double %x)
  %multmp4 = fmul double %calltmp2, %calltmp2
  %addtmp = fadd double %multmp, %multmp4
  ret double %addtmp
}

ready> foo(4.0);
Read top-level expression:
define double @3() {
entry:
  %calltmp = call double @foo(double 4.000000e+00)
  ret double %calltmp
}

Evaluated to 1.000000
```

Whoa, how does the JIT know about sin and cos? The answer is surprisingly simple: in this example, the JIT started execution of a function and got to a function call. It realized that the function was not yet JIT compiled and invoked the standard set of routines to resolve the function. In this case, there is no body defined for the function, so the JIT ended up calling `dlsym("sin")` on the Kaleidoscope process itself. Since `sin` is defined within the JIT's address space, it simply patches up calls in the module to call the libm version of sin directly.

The LLVM JIT provides a number of interfaces (look in the ExecutionEngine.h file) for controlling how unknown functions get resolved. It allows you to establish explicit mappings between IR objects and addresses (useful for LLVM global variables that you want to map to static tables, for example), allows you to dynamically decide on the fly based on the function name, and even allows you to have the JIT compile functions lazily the first time they're called.

One interesting application of this is that we can now extend the language by writing arbitrary C++ code to implement operations. For example, if we add:

```
/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}
```

Now we can produce simple output to the console by using things like: `extern putchard(x); putchard(120);`, which prints a lowercase `x` on the console (120 is the ASCII code for `x`). Similar code could be used to implement file I/O, console input, and many other capabilities in Kaleidoscope.

This completes the JIT and optimizer chapter of the Kaleidoscope tutorial. At this point, we can compile a non-Turing-

complete programming language, optimize and JIT compile it in a user-driven way. Next up we'll look into extending the language with control flow constructs, tackling some interesting LLVM IR issues along the way.

Control Flow

Welcome to Chapter 5 of the "Implementing a language with LLVM" tutorial. Parts 1-4 described the implementation of the simple Kaleidoscope language and included support for generating LLVM IR, followed by optimizations and a JIT compiler. Unfortunately, as presented, Kaleidoscope is mostly useless: it has no control flow other than call and return. This means that you can't have conditional branches in the code, significantly limiting its power. In this episode of "build that compiler", we'll extend Kaleidoscope to have an if/then/else expression plus a simple `for` loop.

5.1 If/Then/Else

Extending Kaleidoscope to support if/then/else is quite straightforward. It basically requires adding support for this "new" concept to the lexer, parser, AST, and LLVM code emitter. This example is nice, because it shows how easy it is to "grow" a language over time, incrementally extending it as new ideas are discovered.

Before we get going on "how" we add this extension, lets talk about "what" we want. The basic idea is that we want to be able to write this sort of thing:

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);
```

In Kaleidoscope, every construct is an expression: there are no statements. As such, the if/then/else expression needs to return a value like any other. Since we're using a mostly functional form, we'll have it evaluate its conditional, then return the `then` or `else` value based on how the condition was resolved. This is very similar to the C "?:" expression.

The semantics of the if/then/else expression is that it evaluates the condition to a boolean equality value: 0.0 is considered to be false and everything else is considered to be true. If the condition is true, the first subexpression is evaluated and returned, if the condition is false, the second subexpression is evaluated and returned. Since Kaleidoscope allows side-effects, this behavior is important to nail down.

Now that we know what we "want", lets break this down into its constituent pieces.

5.1.1 Lexer Extensions for If/Then/Else

The lexer extensions are straightforward. First we add new enum values for the relevant tokens:

```
// control
tok_if = -6, tok_then = -7, tok_else = -8,
Once we have that, we recognize the new keywords in the lexer. This is pretty simple stuff:

...
if (IdentifierStr == "def") return tok_def;
if (IdentifierStr == "extern") return tok_extern;
if (IdentifierStr == "if") return tok_if;
if (IdentifierStr == "then") return tok_then;
if (IdentifierStr == "else") return tok_else;
return tok_identifier;
```

5.1.2 AST Extensions for If/Then/Else

To represent the new expression we add a new AST node for it:

```
/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
    : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};
```

The AST node just has pointers to the various subexpressions.

5.1.3 Parser Extensions for If/Then/Else

Now that we have the relevant tokens coming from the lexer and we have the AST node to build, our parsing logic is relatively straightforward. First we define a new parsing function:

```
/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}
```

Next we hook it up as a primary expression:

```
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  }
}
```

5.1.4 LLVM IR for If/Then/Else

Now that we have it parsing and building the AST, the final piece is adding LLVM code generation support. This is the most interesting part of the if/then/else example, because this is where it starts to introduce new concepts. All of the code above has been thoroughly described in previous chapters.

To motivate the code we want to produce, lets take a look at a simple example. Consider:

```
extern foo();
extern bar();
def baz(x) if x then foo() else bar();
If you disable optimizations, the code you'll (soon) get from Kaleidoscope looks like this:

declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

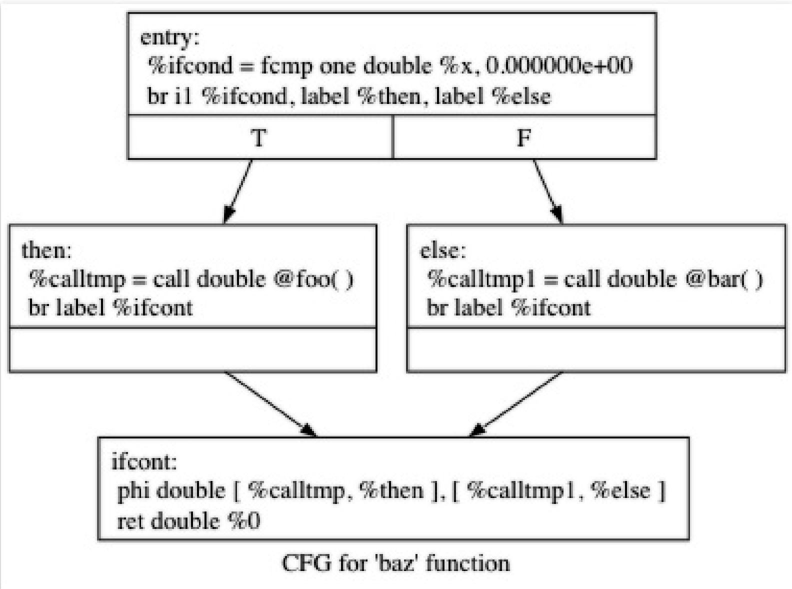
then:       ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont

else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
}
```

To visualize the control flow graph, you can use a nifty feature of the LLVM 'opt' tool. If you put this LLVM IR into "t.ll" and run `llvm-as < t.ll | opt -analyze -view-cfg` , a window will pop up and you'll see this graph:

Example CFG



Example CFG

Another way to get this is to call `F->viewCFG()` or `F->viewCFGOnly()` (where F is a `Function*`) either by inserting actual calls into the code and recompiling or by calling these in the debugger. LLVM has many nice features for visualizing various graphs.

Getting back to the generated code, it is fairly simple: the entry block evaluates the conditional expression (`x` in our case here) and compares the result to 0.0 with the `fcmp one` instruction ('one' is "Ordered and Not Equal"). Based on the result of this expression, the code jumps to either the `then` or `else` blocks, which contain the expressions for the true/false cases.

Once the then/else blocks are finished executing, they both branch back to the `ifcont` block to execute the code that happens after the if/then/else. In this case the only thing left to do is to return to the caller of the function. The question then becomes: how does the code know which expression to return?

The answer to this question involves an important SSA operation: the Phi operation. If you're not familiar with SSA, the wikipedia article is a good introduction and there are various other introductions to it available on your favorite search engine. The short version is that "execution" of the Phi operation requires "remembering" which block control came from. The Phi operation takes on the value corresponding to the input control block. In this case, if control comes in from the `then` block, it gets the value of `calltmp`. If control comes from the `else` block, it gets the value of `calltmp1`.

At this point, you are probably starting to think "Oh no! This means my simple and elegant front-end will have to start generating SSA form in order to use LLVM!". Fortunately, this is not the case, and we strongly advise not implementing an SSA construction algorithm in your front-end unless there is an amazingly good reason to do so. In practice, there are two sorts of values that float around in code written for your average imperative programming language that might need Phi nodes:

Code that involves user variables:

- x = 1; x = x + 1;
- Values that are implicit in the structure of your AST, such as the Phi node in this case.

In Chapter 7 of this tutorial ("mutable variables"), we'll talk about #1 in depth. For now, just believe me that you don't need SSA construction to handle this case. For #2, you have the choice of using the techniques that we will describe for #1, or you can insert Phi nodes directly, if convenient. In this case, it is really really easy to generate the Phi node, so we choose to do it directly.

Okay, enough of the motivation and overview, lets generate code!

5.1.5 Code Generation for If/Then/Else

In order to generate code for this, we implement the Codegen method for IfExprAST:

```
Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                              ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                "ifcond");
This code is straightforward and similar to what we saw before. We emit the expression for the condition, then compare that value to

Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create blocks for the then and else cases.  Insert the 'then' block at the
// end of the function.
BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

Builder.CreateCondBr(CondV, ThenBB, ElseBB);
```

This code creates the basic blocks that are related to the if/then/else statement, and correspond directly to the blocks in the example above. The first line gets the current Function object that is being built. It gets this by asking the builder for the current BasicBlock, and asking that block for its "parent" (the function it is currently embedded into).

Once it has that, it creates three blocks. Note that it passes "TheFunction" into the constructor for the `then` block. This causes the constructor to automatically insert the new block into the end of the specified function. The other two blocks are created, but aren't yet inserted into the function.

Once the blocks are created, we can emit the conditional branch that chooses between them. Note that creating new blocks does not implicitly affect the IRBuilder, so it is still inserting into the block that the condition went into. Also note that it is creating a branch to the `then` block and the `else` block, even though the `else` block isn't inserted into the function yet. This is all ok: it is the standard way that LLVM supports forward references.

```
// Emit then value.
Builder.SetInsertPoint(ThenBB);

Value *ThenV = Then->Codegen();
if (ThenV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Then' can change the current block, update ThenBB for the PHI.
ThenBB = Builder.GetInsertBlock();
```

After the conditional branch is inserted, we move the builder to start inserting into the `then` block. Strictly speaking, this call moves the insertion point to be at the end of the specified block. However, since the `then` block is empty, it also starts out by inserting at the beginning of the block. :)

Once the insertion point is set, we recursively codegen the `then` expression from the AST. To finish off the `then` block, we create an unconditional branch to the merge block. One interesting (and very important) aspect of the LLVM IR is that it requires all basic blocks to be "terminated" with a control flow instruction such as return or branch. This means that all control flow, including fall throughs must be made explicit in the LLVM IR. If you violate this rule, the verifier will emit an error.

The final line here is quite subtle, but is very important. The basic issue is that when we create the Phi node in the merge block, we need to set up the block/value pairs that indicate how the Phi will work. Importantly, the Phi node expects to have an entry for each predecessor of the block in the CFG. Why then, are we getting the current block when we just set it to ThenBB 5 lines above? The problem is that the `then` expression may actually itself change the block that the Builder is emitting into if, for example, it contains a nested "if/then/else" expression. Because calling Codegen recursively could arbitrarily change the notion of the current block, we are required to get an up-to-date value for code that will set up the Phi node.

```
// Emit else block.
TheFunction->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseV = Else->Codegen();
if (ElseV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Else' can change the current block, update ElseBB for the PHI.
ElseBB = Builder.GetInsertBlock();
Code generation for the 'else' block is basically identical to codegen for the 'then' block. The only significant difference is the first line

  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}
```

The first two lines here are now familiar: the first adds the "merge" block to the Function object (it was previously floating, like the else block above). The second block changes the insertion point so that newly created code will go into the "merge" block. Once that is done, we need to create the PHI node and set up the block/value pairs for the PHI.

Finally, the CodeGen function returns the phi node as the value computed by the if/then/else expression. In our example above, this returned value will feed into the code for the top-level function, which will create the return instruction.

Overall, we now have the ability to execute conditional code in Kaleidoscope. With this extension, Kaleidoscope is a fairly complete language that can calculate a wide variety of numeric functions. Next up we'll add another useful expression that is familiar from non-functional languages...

5.2 `for` Loop Expression

Now that we know how to add basic control flow constructs to the language, we have the tools to add more powerful things. Lets add something more aggressive, a `for` expression:

```
extern putchard(char)
def printstar(n)
  for i = 1, i < n, 1.0 in
    putchard(42);  # ascii 42 = '*'

# print 100 '*' characters
printstar(100);
```

This expression defines a new variable (`i` in this case) which iterates from a starting value, while the condition (`i < n` in this case) is true, incrementing by an optional step value (`1.0` in this case). If the step value is omitted, it defaults to `1.0`. While the loop is true, it executes its body expression. Because we don't have anything better to return, we'll just define the loop as always returning `0.0`. In the future when we have mutable variables, it will get more useful.

As before, lets talk about the changes that we need to Kaleidoscope to support this.

5.2.1 Lexer Extensions for the `for` Loop

The lexer extensions are the same sort of thing as for if/then/else:

```
... in enum Token ...
// control
tok_if = -6, tok_then = -7, tok_else = -8,
tok_for = -9, tok_in = -10

... in gettok ...
if (IdentifierStr == "def") return tok_def;
if (IdentifierStr == "extern") return tok_extern;
if (IdentifierStr == "if") return tok_if;
if (IdentifierStr == "then") return tok_then;
if (IdentifierStr == "else") return tok_else;
if (IdentifierStr == "for") return tok_for;
if (IdentifierStr == "in") return tok_in;
return tok_identifier;
```

5.2.2 AST Extensions for the `for` Loop

The AST node is just as simple. It basically boils down to capturing the variable name and the constituent expressions in the node.

```
/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};
```

5.2.3 Parser Extensions for the `for` Loop

The parser code is also fairly standard. The only interesting thing here is handling of the optional step value. The parser

code handles it by checking to see if the second comma is present. If not, it sets the step value to null in the AST node:

```
/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
    return Error("expected '=' after for");
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}
```

5.2.4 LLVM IR for the `for` Loop

Now we get to the good part: the LLVM IR we want to generate for this thing. With the simple example above, we get this LLVM IR (note that this dump is generated with optimizations disabled for clarity):

```
declare double @putchard(double)

define double @printstar(double %n) {
entry:
  ; initial value = 1.0 (inlined into phi)
  br label %loop

loop:       ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  ; body
  %calltmp = call double @putchard(double 4.200000e+01)
  ; increment
  %nextvar = fadd double %i, 1.000000e+00

  ; termination test
  %cmptmp = fcmp ult double %i, %n
  %booltmp = uitofp i1 %cmptmp to double
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop

afterloop:      ; preds = %loop
  ; loop always returns 0.0
  ret double 0.000000e+00
}
```

This loop contains all the same constructs we saw before: a phi node, several expressions, and some basic blocks. Lets see how this fits together.

5.2.5 Code Generation for the `for` Loop

The first part of Codegen is very simple: we just output the start expression for the loop value:

```
Value *ForExprAST::Codegen() {
  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->Codegen();
  if (StartVal == 0) return 0;
```

With this out of the way, the next step is to set up the LLVM basic block for the start of the loop body. In the case above, the whole loop body is one block, but remember that the body code itself could consist of multiple blocks (e.g. if it contains an if/then/else or a for/in expression).

```
// Make the new basic block for the loop header, inserting after current
// block.
Function *TheFunction = Builder.GetInsertBlock()->getParent();
BasicBlock *PreheaderBB = Builder.GetInsertBlock();
BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

// Insert an explicit fall through from the current block to the LoopBB.
Builder.CreateBr(LoopBB);
```

This code is similar to what we saw for if/then/else. Because we will need it to create the Phi node, we remember the block that falls through into the loop. Once we have that, we create the actual block that starts the loop and create an unconditional branch for the fall-through between the two blocks.

```
// Start insertion in LoopBB.
Builder.SetInsertPoint(LoopBB);

// Start the PHI node with an entry for Start.
PHINode *Variable = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2, VarName.c_str());
Variable->addIncoming(StartVal, PreheaderBB);
```

Now that the "preheader" for the loop is set up, we switch to emitting code for the loop body. To begin with, we move the insertion point and create the PHI node for the loop induction variable. Since we already know the incoming value for the

starting value, we add it to the Phi node. Note that the Phi will eventually get a second value for the backedge, but we can't set it up yet (because it doesn't exist!).

```
// Within the loop, the variable is defined equal to the PHI node.  If it
// shadows an existing variable, we have to restore it, so save it now.
Value *OldVal = NamedValues[VarName];
NamedValues[VarName] = Variable;

// Emit the body of the loop.  This, like any other expr, can change the
// current BB.  Note that we ignore the value computed by the body, but don't
// allow an error.
if (Body->Codegen() == 0)
  return 0;
```

Now the code starts to get more interesting. Our 'for' loop introduces a new variable to the symbol table. This means that our symbol table can now contain either function arguments or loop variables. To handle this, before we codegen the body of the loop, we add the loop variable as the current value for its name. Note that it is possible that there is a variable of the same name in the outer scope. It would be easy to make this an error (emit an error and return null if there is already an entry for VarName) but we choose to allow shadowing of variables. In order to handle this correctly, we remember the Value that we are potentially shadowing in OldVal (which will be null if there is no shadowed variable).

Once the loop variable is set into the symbol table, the code recursively codegen's the body. This allows the body to use the loop variable: any references to it will naturally find it in the symbol table.

```
// Emit the step value.
Value *StepVal;
if (Step) {
  StepVal = Step->Codegen();
  if (StepVal == 0) return 0;
} else {
  // If not specified, use 1.0.
  StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
}

Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");
```

Now that the body is emitted, we compute the next value of the iteration variable by adding the step value, or 1.0 if it isn't present. `NextVar` will be the value of the loop variable on the next iteration of the loop.

```
// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Convert condition to a bool by comparing equal to 0.0.
EndCond = Builder.CreateFCmpONE(EndCond,
                      ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                        "loopcond");
```

Finally, we evaluate the exit value of the loop, to determine whether the loop should exit. This mirrors the condition evaluation for the if/then/else statement.

```
// Create the "after loop" block and insert it.
BasicBlock *LoopEndBB = Builder.GetInsertBlock();
BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

// Insert the conditional branch into the end of LoopEndBB.
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

// Any new code will be inserted in AfterBB.
Builder.SetInsertPoint(AfterBB);
```

With the code for the body of the loop complete, we just need to finish up the control flow for it. This code remembers the end block (for the phi node), then creates the block for the loop exit (`afterloop`). Based on the value of the exit condition, it

creates a conditional branch that chooses between executing the loop again and exiting the loop. Any future code is emitted in the `afterloop` block, so it sets the insertion position to it.

```
  // Add a new entry to the PHI node for the backedge.
  Variable->addIncoming(NextVar, LoopEndBB);

  // Restore the unshadowed variable.
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);

  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}
```

The final code handles various cleanups: now that we have the `NextVar` value, we can add the incoming value to the loop PHI node. After that, we remove the loop variable from the symbol table, so that it isn't in scope after the for loop. Finally, code generation of the for loop always returns 0.0, so that is what we return from `ForExprAST::Codegen`.

With this, we conclude the "adding control flow to Kaleidoscope" chapter of the tutorial. In this chapter we added two control flow constructs, and used them to motivate a couple of aspects of the LLVM IR that are important for front-end implementors to know. In the next chapter of our saga, we will get a bit crazier and add user-defined operators to our poor innocent language.

User-defined Operations

Welcome to Chapter 6 of the “Implementing a language with LLVM” tutorial. At this point in our tutorial, we now have a fully functional language that is fairly minimal, but also useful. There is still one big problem with it, however. Our language doesn’t have many useful operators (like division, logical negation, or even any comparisons besides less-than).

This chapter of the tutorial takes a wild digression into adding user-defined operators to the simple and beautiful Kaleidoscope language. This digression now gives us a simple and ugly language in some ways, but also a powerful one at the same time. One of the great things about creating your own language is that you get to decide what is good or bad. In this tutorial we’ll assume that it is okay to use this as a way to show some interesting parsing techniques.

At the end of this tutorial, we’ll run through an example Kaleidoscope application that renders the Mandelbrot set. This gives an example of what you can build with Kaleidoscope and its feature set.

6.1 User-defined Operators: the Idea

The “operator overloading” that we will add to Kaleidoscope is more general than languages like C++. In C++, you are only allowed to redefine existing operators: you can’t programatically change the grammar, introduce new operators, change precedence levels, etc. In this chapter, we will add this capability to Kaleidoscope, which will let the user round out the set of operators that are supported.

The point of going into user-defined operators in a tutorial like this is to show the power and flexibility of using a hand-written parser. Thus far, the parser we have been implementing uses recursive descent for most parts of the grammar and operator precedence parsing for the expressions. See Chapter 2 for details. Without using operator precedence parsing, it would be very difficult to allow the programmer to introduce new operators into the grammar: the grammar is dynamically extensible as the JIT runs.

The two specific features we’ll add are programmable unary operators (right now, Kaleidoscope has no unary operators at all) as well as binary operators. An example of this is:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

Many languages aspire to being able to implement their standard runtime library in the language itself. In Kaleidoscope, we can implement significant parts of the language in the library!

We will break down implementation of these features into two parts: implementing support for user-defined binary operators and adding unary operators.

6.2 User-defined Binary Operators

Adding support for user-defined binary operators is pretty simple with our current framework. We'll first add support for the unary/binary keywords:

```
enum Token {
  ...
  // operators
  tok_binary = -11, tok_unary = -12
};
...
static int gettok() {
...
    if (IdentifierStr == "for") return tok_for;
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    return tok_identifier;
```

This just adds lexer support for the unary and binary keywords, like we did in previous chapters. One nice thing about our current AST, is that we represent binary operators with full generalisation by using their ASCII code as the opcode. For our extended operators, we'll use this same representation, so we don't need any new AST or parser support.

On the other hand, we have to be able to represent the definitions of these new operators, in the "def binary| 5" part of the function definition. In our grammar so far, the "name" for the function definition is parsed as the "prototype" production and into the PrototypeAST AST node. To represent our new user-defined operators as prototypes, we have to extend the PrototypeAST AST node like this:

```
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its argument names as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool isOperator;
  unsigned Precedence;  // Precedence if a binary op.
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args,
               bool isoperator = false, unsigned prec = 0)
  : Name(name), Args(args), isOperator(isoperator), Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Args.size() == 1; }
  bool isBinaryOp() const { return isOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size()-1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }

  Function *Codegen();
};
```

Basically, in addition to knowing a name for the prototype, we now keep track of whether it was an operator, and if it was, what precedence level the operator is at. The precedence is only used for binary operators (as you'll see below, it just doesn't apply for unary operators). Now that we have a way to represent the prototype for a user-defined operator, we need to parse it:

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return ErrorP("Invalid precedecnce: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }

  if (CurTok != '(')
    return ErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return ErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return ErrorP("Invalid number of operands for operator");

  return new PrototypeAST(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}
```

This is all fairly straightforward parsing code, and we have already seen a lot of similar code in the past. One interesting part about the code above is the couple lines that set up FnName for binary operators. This builds names like “binary@” for a newly defined “@” operator. This then takes advantage of the fact that symbol names in the LLVM symbol table are allowed to have any character in them, including embedded nul characters.

The next interesting thing to add, is codegen support for these binary operators. Given our current structure, this is a simple addition of a default case for our existing binary operator node:

```
Value *BinaryExprAST::Codegen() {
  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = TheModule->getFunction(std::string("binary")+Op);
  assert(F && "binary operator not found!");

  Value *Ops[2] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}
```

As you can see above, the new code is actually really simple. It just does a lookup for the appropriate operator in the symbol table and generates a function call to it. Since user-defined operators are just built as normal functions (because the “prototype” boils down to a function with the right name) everything falls into place.

The final piece of code we are missing, is a bit of top-level magic:

```
Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // If this is an operator, install it.
  if (Proto->isBinaryOp())
    BinopPrecedence[Proto->getOperatorName()] = Proto->getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  if (Value *RetVal = Body->Codegen()) {
    ...
```

Basically, before codegening a function, if it is a user-defined operator, we register it in the precedence table. This allows the binary operator parsing logic we already have in place to handle it. Since we are working on a fully-general operator precedence parser, this is all we need to do to “extend the grammar”.

Now we have useful user-defined binary operators. This builds a lot on the previous framework we built for other operators. Adding unary operators is a bit more challenging, because we don’t have any framework for it yet - lets see what it takes.

6.3 User-defined Unary Operators

Since we don’t currently support unary operators in the Kaleidoscope language, we’ll need to add everything to support them. Above, we added simple support for the ‘unary’ keyword to the lexer. In addition to that, we need an AST node:

```
/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
  char Opcode;
  ExprAST *Operand;
public:
  UnaryExprAST(char opcode, ExprAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};
```

This AST node is very simple and obvious by now. It directly mirrors the binary operator AST node, except that it only has one child. With this, we need to add the parsing logic. Parsing a unary operator is pretty simple: we'll add a new function to do it:

```
/// unary
///   ::= primary
///   ::= '!' unary
static ExprAST *ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (ExprAST *Operand = ParseUnary())
    return new UnaryExprAST(Opc, Operand);
  return 0;
}
```

The grammar we add is pretty straightforward here. If we see a unary operator when parsing a primary operator, we eat the operator as a prefix and parse the remaining piece as another unary operator. This allows us to handle multiple unary operators (e.g. `!!x`). Note that unary operators can't have ambiguous parses like binary operators can, so there is no need for precedence information.

The problem with this function, is that we need to call ParseUnary from somewhere. To do this, we change previous callers of ParsePrimary to call ParseUnary instead:

```
/// binoprhs
///   ::= ('+' unary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  ...
    // Parse the unary expression after the binary operator.
    ExprAST *RHS = ParseUnary();
    if (!RHS) return 0;
  ...
}
/// expression
///   ::= unary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParseUnary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}
```

With these two simple changes, we are now able to parse unary operators and build the AST for them. Next up, we need to add parser support for prototypes, to parse the unary operator prototype. We extend the binary operator code above with:

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return ErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    ...
```

As with binary operators, we name unary operators with a name that includes the operator character. This assists us at code generation time. Speaking of, the final piece we need to add is codegen support for unary operators. It looks like this:

```
Value *UnaryExprAST::Codegen() {
  Value *OperandV = Operand->Codegen();
  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opcode);
  if (F == 0)
    return ErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}
```

This code is similar to, but simpler than, the code for binary operators. It is simpler primarily because it doesn't need to handle any predefined operators.

6.4 Kicking the Tires

It is somewhat hard to believe, but with a few simple extensions we've covered in the last chapters, we have grown a real-ish language. With this, we can do a lot of interesting things, including I/O, math, and a bunch of other things. For example, we can now add a nice sequencing operator (printd is defined to print out the specified value and a newline):

```
ready> extern printd(x);
Read extern:
declare double @printd(double)

ready> def binary : 1 (x y) 0;  # Low-precedence operator that ignores operands.
..
ready> printd(123) : printd(456) : printd(789);
123.000000
456.000000
789.000000
Evaluated to 0.000000
```

We can also define a bunch of other "primitive" operations, such as:

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Unary negate.
def unary-(v)
  0-v;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary logical or, which does not short circuit.
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Binary logical and, which does not short circuit.
def binary& 6 (LHS RHS)
  if !LHS then
    0
  else
    !!RHS;

# Define = with slightly lower precedence than relationals.
def binary = 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;
```

Given the previous if/then/else support, we can also define interesting functions for I/O. For example, the following prints out a character whose “density” reflects the value passed in: the lower the value, the denser the character:

```
ready>

extern putchard(char)
def printdensity(d)
  if d > 8 then
    putchard(32)  # ' '
  else if d > 4 then
    putchard(46)  # '.'
  else if d > 2 then
    putchard(43)  # '+'
  else
    putchard(42); # '*'
...
ready> printdensity(1): printdensity(2): printdensity(3):
       printdensity(4): printdensity(5): printdensity(9):
       putchard(10);
**++.
```

Evaluated to 0.000000 Based on these simple primitive operations, we can start to define more interesting things. For example, here’s a little function that solves for the number of iterations it takes a function in the complex plane to converge:

```
# Determine whether the specific location diverges.
# Solve for z = z^2 + c in the complex plane.
def mandleconverger(real imag iters creal cimag)
  if iters > 255 | (real*real + imag*imag > 4) then
    iters
  else
    mandleconverger(real*real - imag*imag + creal,
                    2*real*imag + cimag,
                    iters+1, creal, cimag);

# Return the number of iterations required for the iteration to escape
def mandleconverge(real imag)
  mandleconverger(real, imag, 0, real, imag);
```

This `z = z2 + c` function is a beautiful little creature that is the basis for computation of the Mandelbrot Set. Our mandelconverge function returns the number of iterations that it takes for a complex orbit to escape, saturating to 255. This is not a very useful function by itself, but if you plot its value over a two-dimensional plane, you can see the Mandelbrot set. Given that we are limited to using putchard here, our amazing graphical output is limited, but we can whip together something using the density plotter above:

```
# Compute and plot the mandlebrot set with the specified 2 dimensional range
# info.
def mandelhelp(xmin xmax xstep   ymin ymax ystep)
  for y = ymin, y < ymax, ystep in (
    (for x = xmin, x < xmax, xstep in
       printdensity(mandleconverge(x,y)))
    : putchard(10)
  )

# mandel - This is a convenient helper function for plotting the mandelbrot set
# from the specified position with the specified Magnification.
def mandel(realstart imagstart realmag imagmag)
  mandelhelp(realstart, realstart+realmag*78, realmag,
             imagstart, imagstart+imagmag*40, imagmag);
```

Given this, we can try plotting out the mandlebrot set! Lets try it out:

```
ready> mandel(-2.3, -1.3, 0.05, 0.07);
*******************************+++++++++++*************************************
*************************+++++++++++++++++++++++*******************************
**********************+++++++++++++++++++++++++++++****************************
*******************+++++++++++++++++++++.. ...++++++++*************************
*****************++++++++++++++++++++++.... ...+++++++++***********************
***************+++++++++++++++++++++++.....   ...+++++++++*********************
**************+++++++++++++++++++++++....     ....+++++++++********************
*************++++++++++++++++++++++......      .....++++++++*******************
************+++++++++++++++++++++.......       .......+++++++******************
***********+++++++++++++++++++....                ... .+++++++*****************
**********+++++++++++++++++.......                     .+++++++****************
*********++++++++++++++...........                    ...+++++++***************
********++++++++++++............                      ...++++++++**************
********++++++++++... ..........                        .++++++++**************
*******+++++++++.....                                   .+++++++++*************
*******++++++++......                                  ..+++++++++*************
*******++++++.......                                   ..+++++++++*************
*******+++++......                                     ..+++++++++*************
*******.... ....                                      ...+++++++++*************
*******.... .                                         ...+++++++++*************
*******+++++......                                    ...+++++++++*************
*******++++++.......                                   ..+++++++++*************
*******++++++++......                                   .+++++++++*************
*******+++++++++.....                                  ..+++++++++*************
********++++++++++... ..........                        .++++++++**************
********++++++++++++............                      ...++++++++**************
*********++++++++++++++..........                     ...+++++++***************
**********++++++++++++++++........                     .+++++++****************
**********++++++++++++++++++++....                ... ..+++++++****************
***********++++++++++++++++++++++.......       .......++++++++*****************
************+++++++++++++++++++++++......      ......++++++++******************
**************+++++++++++++++++++++++....      ....++++++++********************
***************+++++++++++++++++++++++.....   ...+++++++++*********************
*****************++++++++++++++++++++++....  ...++++++++***********************
*******************+++++++++++++++++++++......++++++++*************************
```

```
+++++++++++++++++++++.....++++++++
*********************++++++++++++++++++++++.++++++++***************************
*************************+++++++++++++++++++++++*******************************
******************************+++++++++++++************************************
*******************************************************************************
*******************************************************************************
*******************************************************************************
Evaluated to 0.000000
ready> mandel(-2, -1, 0.02, 0.04);
**************************+++++++++++++++++++++++++++++++++++++++++++++++++++++
***********************++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*********************+++++++++++++++++++++++++++++++++++++++++++++++++++++++++.
*******************+++++++++++++++++++++++++++++++++++++++++++++++++++++++++...
*****************+++++++++++++++++++++++++++++++++++++++++++++++++++++++++.....
***************++++++++++++++++++++++++++++++++++++++++++++++++++++++++........
**************++++++++++++++++++++++++++++++++++++++++++++++++++++++...........
************+++++++++++++++++++++++++++++++++++++++++++++++++++++..............
***********++++++++++++++++++++++++++++++++++++++++++++++++++........         .
**********++++++++++++++++++++++++++++++++++++++++++++++.............
********+++++++++++++++++++++++++++++++++++++++++++..................
*******+++++++++++++++++++++++++++++++++++++++.......................
******+++++++++++++++++++++++++++++++++++...........................
*****++++++++++++++++++++++++++++++++............................
*****++++++++++++++++++++++++++++..............................
****++++++++++++++++++++++++++......   ........................
***++++++++++++++++++++++++.........     ......    ...........
***++++++++++++++++++++++............
**+++++++++++++++++++++..............
**+++++++++++++++++++................
*++++++++++++++++++.................
*++++++++++++++++............ ...
*++++++++++++++..............
*+++....++++................
*..........  ...........
*
*..........  ...........
*+++....++++................
*++++++++++++++..............
*++++++++++++++++............ ...
*++++++++++++++++++.................
**+++++++++++++++++++................
**+++++++++++++++++++++..............
***++++++++++++++++++++++............
***++++++++++++++++++++++++.........     ......    ...........
****++++++++++++++++++++++++++......   ........................
*****++++++++++++++++++++++++++++..............................
*****++++++++++++++++++++++++++++++++............................
******+++++++++++++++++++++++++++++++++++...........................
*******+++++++++++++++++++++++++++++++++++++++.......................
********+++++++++++++++++++++++++++++++++++++++++++..................
Evaluated to 0.000000
ready> mandel(-0.9, -1.4, 0.02, 0.03);
*******************************************************************************
*******************************************************************************
*******************************************************************************
**********+++++++++++++++++++++************************************************
*+++++++++++++++++++++++++++++++++++++++***************************************
+++++++++++++++++++++++++++++++++++++++++++++**********************************
++++++++++++++++++++++++++++++++++++++++++++++++++*****************************
++++++++++++++++++++++++++++++++++++++++++++++++++++++*************************
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++**********************
+++++++++++++++++++++++++++++++++.........++++++++++++++++++*******************
+++++++++++++++++++++++++++++++....   ......+++++++++++++++++++****************
+++++++++++++++++++++++++++++.......  ........+++++++++++++++++++**************
++++++++++++++++++++++++++++........   ........++++++++++++++++++++************
+++++++++++++++++++++++++++.........     ..  ...+++++++++++++++++++++**********
++++++++++++++++++++++++++...........        ....++++++++++++++++++++++********
++++++++++++++++++++++++.............       .......++++++++++++++++++++++******
+++++++++++++++++++++++.............        ........+++++++++++++++++++++++****
++++++++++++++++++++++...........           ..........++++++++++++++++++++++***
++++++++++++++++++++...........                .........++++++++++++++++++++++*
++++++++++++++++++............                 ...........++++++++++++++++++++
++++++++++++++++...............                .............++++++++++++++++++
++++++++++++++.................                ...............++++++++++++++++
++++++++++++..................                 .................++++++++++++++
+++++++++..................                     .................+++++++++++++
++++++........        .                              .........  ..++++++++++++
++............                                        ......    ....++++++++++
.............                                                    ...++++++++++
.............                                                    ....+++++++++
.............                                                    .....++++++++
............                                                     ......++++++++
..........                                                       .......++++++++
```

```
.........                         ........+++++++
.........                         ........+++++++
.........                             ....+++++++
........                               ...+++++++
.......                                ...+++++++
                                          ....+++++++
                                         .....+++++++
                                          ....+++++++
                                          ....+++++++
                                          ....+++++++
Evaluated to 0.000000
ready> ^D
```

At this point, you may be starting to realize that Kaleidoscope is a real and powerful language. It may not be self-similar :), but it can be used to plot things that are!

With this, we conclude the “adding user-defined operators” chapter of the tutorial. We have successfully augmented our language, adding the ability to extend the language in the library, and we have shown how this can be used to build a simple but interesting end-user application in Kaleidoscope. At this point, Kaleidoscope can build a variety of applications that are functional and can call functions with side-effects, but it can’t actually define and mutate a variable itself.

Strikingly, variable mutation is an important feature of some languages, and it is not at all obvious how to add support for mutable variables without having to add an “SSA construction” phase to your front-end. In the next chapter, we will describe how you can add variable mutation without building SSA in your front-end.

Mutable Variables

7.1 Chapter 7 Introduction

Welcome to Chapter 7 of the “Implementing a language with LLVM” tutorial. In chapters 1 through 6, we’ve built a very respectable, albeit simple, functional programming language. In our journey, we learned some parsing techniques, how to build and represent an AST, how to build LLVM IR, and how to optimize the resultant code as well as JIT compile it.

While Kaleidoscope is interesting as a functional language, the fact that it is functional makes it “too easy” to generate LLVM IR for it. In particular, a functional language makes it very easy to build LLVM IR directly in SSA form. Since LLVM requires that the input code be in SSA form, this is a very nice property and it is often unclear to newcomers how to generate code for an imperative language with mutable variables.

The short (and happy) summary of this chapter is that there is no need for your front-end to build SSA form: LLVM provides highly tuned and well tested support for this, though the way it works is a bit unexpected for some.

7.2 Why is this a hard problem?

To understand why mutable variables cause complexities in SSA construction, consider this extremely simple C example:

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
```

In this case, we have the variable “X”, whose value depends on the path executed in the program. Because there are two different possible values for X before the return instruction, a PHI node is inserted to merge the two values. The LLVM IR that we want for this example looks like this:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
```

In this example, the loads from the G and H global variables are explicit in the LLVM IR, and they live in the then/else branches of the if statement (cond_true/cond_false). In order to merge the incoming values, the `X.2` phi node in the `cond_next` block selects the right value to use based on where control flow is coming from: if control flow comes from the `cond_false` block, `X.2` gets the value of `X.1`. Alternatively, if control flow comes from cond_true, it gets the value of `X.0`. The intent of this chapter is not to explain the details of SSA form. For more information, see one of the many online references.

The question for this article is “who places the phi nodes when lowering assignments to mutable variables?”. The issue here is that LLVM requires that its IR be in SSA form: there is no “non-ssa” mode for it. However, SSA construction requires non-trivial algorithms and data structures, so it is inconvenient and wasteful for every front-end to have to reproduce this logic.

7.3 Memory in LLVM

The ‘trick’ here is that while LLVM does require all register values to be in SSA form, it does not require (or permit) memory objects to be in SSA form. In the example above, note that the loads from G and H are direct accesses to G and H: they are not renamed or versioned. This differs from some other compiler systems, which do try to version memory objects. In LLVM, instead of encoding dataflow analysis of memory into the LLVM IR, it is handled with Analysis Passes which are computed on demand.

With this in mind, the high-level idea is that we want to make a stack variable (which lives in memory, because it is on the stack) for each mutable object in a function. To take advantage of this trick, we need to talk about how LLVM represents stack variables.

In LLVM, all memory accesses are explicit with load/store instructions, and it is carefully designed not to have (or need) an “address-of” operator. Notice how the type of the @G/@H global variables is actually “i32*” even though the variable is defined as `i32`. What this means is that @G defines space for an i32 in the global data area, but its name actually refers to the address for that space. Stack variables work the same way, except that instead of being declared with global variable definitions, they are declared with the LLVM alloca instruction:

```
define i32 @example() {
entry:
  %X = alloca i32           ; type of %X is i32*.
  ...
  %tmp = load i32* %X       ; load the stack value %X from the stack.
  %tmp2 = add i32 %tmp, 1   ; increment it
  store i32 %tmp2, i32* %X  ; store it back
  ...
```

This code shows an example of how you can declare and manipulate a stack variable in the LLVM IR. Stack memory allocated with the alloca instruction is fully general: you can pass the address of the stack slot to functions, you can store it in other variables, etc. In our example above, we could rewrite the example to use the alloca technique to avoid using a PHI node:

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32* %X       ; Read X
  ret i32 %X.2
}
```

With this, we have discovered a way to handle arbitrary mutable variables without the need to create Phi nodes at all:

Each mutable variable becomes a stack allocation. Each read of the variable becomes a load from the stack. Each update of the variable becomes a store to the stack. Taking the address of a variable just uses the stack address directly. While this solution has solved our immediate problem, it introduced another one: we have now apparently introduced a lot of stack traffic for very simple and common operations, a major performance problem. Fortunately for us, the LLVM optimizer has a highly-tuned optimization pass named “mem2reg” that handles this case, promoting allocas like this into SSA registers, inserting Phi nodes as appropriate. If you run this example through the pass, for example, you’ll get:

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

The mem2reg pass implements the standard “iterated dominance frontier” algorithm for constructing SSA form and has a number of optimizations that speed up (very common) degenerate cases. The mem2reg optimization pass is the answer to dealing with mutable variables, and we highly recommend that you depend on it. Note that mem2reg only works on variables in certain circumstances:

mem2reg is alloca-driven: it looks for allocas and if it can handle them, it promotes them. It does not apply to global variables or heap allocations. mem2reg only looks for alloca instructions in the entry block of the function. Being in the entry block guarantees that the alloca is only executed once, which makes analysis simpler. mem2reg only promotes allocas whose uses are direct loads and stores. If the address of the stack object is passed to a function, or if any funny pointer arithmetic is involved, the alloca will not be promoted. mem2reg only works on allocas of first class values (such as pointers, scalars and vectors), and only if the array size of the allocation is 1 (or missing in the .ll file). mem2reg is not capable of promoting structs or arrays to registers. Note that the “scalarrepl” pass is more powerful and can promote structs, “unions”, and arrays in many cases. All of these properties are easy to satisfy for most imperative languages, and we’ll illustrate it below with Kaleidoscope. The final question you may be asking is: should I bother with this nonsense for my front-end? Wouldn’t it be better if I just did SSA construction directly, avoiding use of the mem2reg optimization pass? In short, we strongly recommend that you use this technique for building SSA form, unless there is an extremely good reason not to. Using this technique is:

Proven and well tested: clang uses this technique for local mutable variables. As such, the most common clients of LLVM are using this to handle a bulk of their variables. You can be sure that bugs are found fast and fixed early. Extremely Fast: mem2reg has a number of special cases that make it fast in common cases as well as fully general. For example, it has fast-paths for variables that are only used in a single block, variables that only have one assignment point, good heuristics to avoid insertion of unneeded phi nodes, etc. Needed for debug info generation: Debug information in LLVM relies on having the address of the variable exposed so that debug info can be attached to it. This technique dovetails very naturally with this style of debug info. If nothing else, this makes it much easier to get your front-end up and running, and is very simple to implement. Lets extend Kaleidoscope with mutable variables now!

7.4 Mutable Variables in Kaleidoscope

Now that we know the sort of problem we want to tackle, lets see what this looks like in the context of our little Kaleidoscope language. We’re going to add two features:

The ability to mutate variables with the `=` operator. The ability to define new variables. While the first item is really what this is about, we only have variables for incoming arguments as well as for induction variables, and redefining those only goes so far :). Also, the ability to define new variables is a useful thing regardless of whether you will be mutating them. Here's a motivating example that shows how we could use these:

```
# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

# Recursive fib, we could do this before.
def fib(x)
  if (x < 3) then
    1
  else
    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :
     a = b :
     b = c) :
  b;

# Call it.
fibi(10);
```

In order to mutate variables, we have to change our existing variables to use the "alloca trick". Once we have that, we'll add our new operator, then extend Kaleidoscope to support new variable definitions.

7.5 Adjusting Existing Variables for Mutation

The symbol table in Kaleidoscope is managed at code generation time by the `NamedValues` map. This map currently keeps track of the LLVM `Value*` that holds the double value for the named variable. In order to support mutation, we need to change this slightly, so that it NamedValues holds the memory location of the variable in question. Note that this change is a refactoring: it changes the structure of the code, but does not (by itself) change the behavior of the compiler. All of these changes are isolated in the Kaleidoscope code generator.

At this point in Kaleidoscope's development, it only supports variables for two things: incoming arguments to functions and the induction variable of `for` loops. For consistency, we'll allow mutation of these variables in addition to other user-defined variables. This means that these will both need memory locations.

To start our transformation of Kaleidoscope, we'll change the NamedValues map so that it maps to `AllocaInst*` instead of `Value*`. Once we do this, the C++ compiler will tell us what parts of the code we need to update:

```
static std::map<std::string, AllocaInst*> NamedValues;
```

Also, since we will need to create these alloca's, we'll use a helper function that ensures that the allocas are created in the entry block of the function:

```
/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                 TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(getGlobalContext()), 0,
                           VarName.c_str());
}
```

This funny looking code creates an IRBuilder object that is pointing at the first instruction (.begin()) of the entry block. It then creates an alloca with the expected name and returns it. Because all values in Kaleidoscope are doubles, there is no need to pass in a type to use.

With this in place, the first functionality change we want to make is to variable references. In our new scheme, variables live on the stack, so code generating a reference to them actually needs to produce a load from the stack slot:

```
Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (V == 0) return ErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```

As you can see, this is pretty straightforward. Now we need to update the things that define the variables to set up the alloca. We'll start with ForExprAST::Codegen (see the full code listing for the unabridged code):

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create an alloca for the variable in the entry block.
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

  // Emit the start code first, without 'variable' in scope.
Value *StartVal = Start->Codegen();
if (StartVal == 0) return 0;

// Store the value into the alloca.
Builder.CreateStore(StartVal, Alloca);
...

// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Reload, increment, and restore the alloca.  This handles the case where
// the body of the loop mutates the variable.
Value *CurVar = Builder.CreateLoad(Alloca);
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);
...
```

This code is virtually identical to the code before we allowed mutable variables. The big difference is that we no longer have to construct a PHI node, and we use load/store to access the variable as needed.

To support mutable argument variables, we need to also make allocas for them. The code for this is also pretty simple:

```
/// CreateArgumentAllocas - Create an alloca for each argument and register the
/// argument in the symbol table so that references to it will succeed.
void PrototypeAST::CreateArgumentAllocas(Function *F) {
  Function::arg_iterator AI = F->arg_begin();
  for (unsigned Idx = 0, e = Args.size(); Idx != e; ++Idx, ++AI) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(F, Args[Idx]);

    // Store the initial value into the alloca.
    Builder.CreateStore(AI, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = Alloca;
  }
}
```

For each argument, we make an alloca, store the input value to the function into the alloca, and register the alloca as the memory location for the argument. This method gets invoked by FunctionAST::Codegen right after it sets up the entry block for the function.

The final missing piece is adding the mem2reg pass, which allows us to get good codegen once again:

```
// Set up the optimizer pipeline.  Start with registering info about how the
// target lays out data structures.
OurFPM.add(new DataLayout(*TheExecutionEngine->getDataLayout()));
// Promote allocas to registers.
OurFPM.add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
// Reassociate expressions.
OurFPM.add(createReassociatePass());
It is interesting to see what the code looks like before and after the mem2reg optimization runs. For example, this is the before/after

define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  br label %ifcont

else:       ; preds = %entry
  %x3 = load double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

Here there is only one variable (x, the input argument) but you can still see the extremely simple-minded code generation strategy we are using. In the entry block, an alloca is created, and the initial input value is stored into it. Each reference to the variable does a reload from the stack. Also, note that we didn't modify the if/then/else expression, so it still inserts a PHI node. While we could make an alloca for it, it is actually easier to create a PHI node for it, so we still just make the PHI.

Here is the code after the mem2reg pass runs:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:
  br label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

This is a trivial case for mem2reg, since there are no redefinitions of the variable. The point of showing this is to calm your tension about inserting such blatent inefficiencies :).

After the rest of the optimizers run, we get:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp ueq double %booltmp, 0.000000e+00
  br i1 %ifcond, label %else, label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  ret double %addtmp

ifcont:
  ret double 1.000000e+00
}
```

Here we see that the simplifycfg pass decided to clone the return instruction into the end of the `else` block. This allowed it to eliminate some branches and the PHI node.

Now that all symbol table references are updated to use stack variables, we'll add the assignment operator.

7.6 New Assignment Operator

With our current framework, adding a new assignment operator is really simple. We will parse it just like any other binary operator, but handle it internally (instead of allowing the user to define it). The first step is to set a precedence:

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

Now that the parser knows the precedence of the binary operator, it takes care of all the parsing and AST generation. We just need to implement codegen for the assignment operator. This looks like:

```
Value *BinaryExprAST::Codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    // Assignment requires the LHS to be an identifier.
    VariableExprAST *LHSE = dynamic_cast<VariableExprAST*>(LHS);
    if (!LHSE)
      return ErrorV("destination of '=' must be a variable");
```

Unlike the rest of the binary operators, our assignment operator doesn't follow the "emit LHS, emit RHS, do computation" model. As such, it is handled as a special case before the other binary operators are handled. The other strange thing is that it requires the LHS to be a variable. It is invalid to have "(x+1) = expr" - only things like "x = expr" are allowed.

```
    // Codegen the RHS.
    Value *Val = RHS->Codegen();
    if (Val == 0) return 0;

    // Look up the name.
    Value *Variable = NamedValues[LHSE->getName()];
    if (Variable == 0) return ErrorV("Unknown variable name");

    Builder.CreateStore(Val, Variable);
    return Val;
  }
  ...
```

Once we have the variable, codegen'ing the assignment is straightforward: we emit the RHS of the assignment, create a store, and return the computed value. Returning a value allows for chained assignments like `X = (Y = Z)`.

Now that we have an assignment operator, we can mutate loop variables and arguments. For example, we can now run code like this:

```
# Function to print a double.
extern printd(x);

# Define ':' for sequencing: as a low-precedence operator that ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

def test(x)
  printd(x) :
  x = 4 :
  printd(x);

test(123);
```

When run, this example prints "123" and then "4", showing that we did actually mutate the value! Okay, we have now officially implemented our goal: getting this to work requires SSA construction in the general case. However, to be really useful, we want the ability to define our own local variables, lets add this next!

7.7 User-defined Local Variables

Adding var/in is just like any other other extensions we made to Kaleidoscope: we extend the lexer, the parser, the AST and the code generator. The first step for adding our new 'var/in' construct is to extend the lexer. As before, this is pretty trivial, the code looks like this:

```
enum Token {
  ...
  // var definition
  tok_var = -13
...
}
...
static int gettok() {
...
    if (IdentifierStr == "in") return tok_in;
    if (IdentifierStr == "binary") return tok_binary;
    if (IdentifierStr == "unary") return tok_unary;
    if (IdentifierStr == "var") return tok_var;
    return tok_identifier;
...
```

The next step is to define the AST node that we will construct. For var/in, it looks like this:

```
/// VarExprAST - Expression class for var/in
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  ExprAST *Body;
public:
  VarExprAST(const std::vector<std::pair<std::string, ExprAST*> > &varnames,
             ExprAST *body)
  : VarNames(varnames), Body(body) {}

  virtual Value *Codegen();
};
```

var/in allows a list of names to be defined all at once, and each name can optionally have an initializer value. As such, we capture this information in the VarNames vector. Also, var/in has a body, this body is allowed to access the variables defined by the var/in.

With this in place, we can define the parser pieces. The first thing we do is add it as a primary expression:

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  case tok_var:        return ParseVarExpr();
  }
}
```

Next we define ParseVarExpr:

```
/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static ExprAST *ParseVarExpr() {
  getNextToken();  // eat the var.

  std::vector<std::pair<std::string, ExprAST*> > VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)
    return Error("expected identifier after var");
The first part of this code parses the list of identifier/expr pairs into the local VarNames vector.

while (1) {
  std::string Name = IdentifierStr;
  getNextToken();  // eat identifier.

  // Read the optional initializer.
  ExprAST *Init = 0;
  if (CurTok == '=') {
    getNextToken(); // eat the '='.

    Init = ParseExpression();
    if (Init == 0) return 0;
  }

  VarNames.push_back(std::make_pair(Name, Init));

  // End of var list, exit loop.
  if (CurTok != ',') break;
  getNextToken(); // eat the ','.

  if (CurTok != tok_identifier)
    return Error("expected identifier list after var");
}
```

Once all the variables are parsed, we then parse the body and create the AST node:

```
  // At this point, we have to have 'in'.
  if (CurTok != tok_in)
    return Error("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new VarExprAST(VarNames, Body);
}
```

Now that we can parse and represent the code, we need to support emission of LLVM IR for it. This code starts out with:

```
Value *VarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second;
```

Basically it loops over all the variables, installing them one at a time. For each variable we put into the symbol table, we remember the previous value that we replace in OldBindings.

```
  // Emit the initializer before adding the variable to scope, this prevents
  // the initializer from referencing the variable itself, and permits stuff
  // like this:
  //  var a = 1 in
  //    var a = a in ...   # refers to outer 'a'.
  Value *InitVal;
  if (Init) {
    InitVal = Init->Codegen();
    if (InitVal == 0) return 0;
  } else { // If not specified, use 0.0.
    InitVal = ConstantFP::get(getGlobalContext(), APFloat(0.0));
  }

  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
  Builder.CreateStore(InitVal, Alloca);

  // Remember the old variable binding so that we can restore the binding when
  // we unrecurse.
  OldBindings.push_back(NamedValues[VarName]);

  // Remember this binding.
  NamedValues[VarName] = Alloca;
}
```

There are more comments here than code. The basic idea is that we emit the initializer, create the alloca, then update the symbol table to point to it. Once all the variables are installed in the symbol table, we evaluate the body of the var/in expression:

```
// Codegen the body, now that all vars are in scope.
Value *BodyVal = Body->Codegen();
if (BodyVal == 0) return 0;
Finally, before returning, we restore the previous variable bindings:

  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}
```

The end result of all of this is that we get properly scoped variable definitions, and we even (trivially) allow mutation of them :).

With this, we completed what we set out to do. Our nice iterative fib example from the intro compiles and runs just fine. The mem2reg pass optimizes all of our stack variables into SSA registers, inserting PHI nodes where needed, and our front-end remains simple: no “iterated dominance frontier” computation anywhere in sight.

Conclusion

Welcome to the final chapter of the “Implementing a language with LLVM” tutorial. In the course of this tutorial, we have grown our little Kaleidoscope language from being a useless toy, to being a semi-interesting (but probably still useless) toy. :)

It is interesting to see how far we’ve come, and how little code it has taken. We built the entire lexer, parser, AST, code generator, and an interactive run-loop (with a JIT!) by-hand in under 700 lines of (non-comment/non-blank) code.

Our little language supports a couple of interesting features: it supports user defined binary and unary operators, it uses JIT compilation for immediate evaluation, and it supports a few control flow constructs with SSA construction.

Part of the idea of this tutorial was to show you how easy and fun it can be to define, build, and play with languages. Building a compiler need not be a scary or mystical process! Now that you’ve seen some of the basics, I strongly encourage you to take the code and hack on it. For example, try adding:

- **global variables** - While global variables have questional value in modern software engineering, they are often useful when putting together quick little hacks like the Kaleidoscope compiler itself. Fortunately, our current setup makes it very easy to add global variables: just have value lookup check to see if an unresolved variable is in the global variable symbol table before rejecting it. To create a new global variable, make an instance of the LLVM GlobalVariable class.
- **typed variables** - Kaleidoscope currently only supports variables of type double. This gives the language a very nice elegance, because only supporting one type means that you never have to specify types. Different languages have different ways of handling this. The easiest way is to require the user to specify types for every variable definition, and record the type of the variable in the symbol table along with its Value*.
- **arrays, structs, vectors, etc** - Once you add types, you can start extending the type system in all sorts of interesting ways. Simple arrays are very easy and are quite useful for many different applications. Adding them is mostly an exercise in learning how the LLVM getelementptr instruction works: it is so nifty/unconventional, it has its own FAQ! If you add support for recursive types (e.g. linked lists), make sure to read the section in the LLVM Programmer’s Manual that describes how to construct them.
- **standard runtime** - Our current language allows the user to access arbitrary external functions, and we use it for things like “printd” and “putchard”. As you extend the language to add higher-level constructs, often these constructs make the most sense if they are lowered to calls into a language-supplied runtime. For example, if you add hash tables to the language, it would probably make sense to add the routines to a runtime, instead of inlining them all the way.
- **memory management** - Currently we can only access the stack in Kaleidoscope. It would also be useful to be able to allocate heap memory, either with calls to the standard libc malloc/free interface or with a garbage collector. If you would like to use garbage collection, note that LLVM fully supports Accurate Garbage Collection including algorithms that move objects and need to scan/update the stack.
- **debugger support** - LLVM supports generation of DWARF Debug info which is understood by common debuggers like GDB. Adding support for debug info is fairly straightforward. The best way to understand it is to compile some C/C++ code with “clang -g -O0” and taking a look at what it produces.
- **exception handling support** - LLVM supports generation of zero cost exceptions which interoperate with code compiled in other languages. You could also generate code by implicitly making every function return an error value and checking it. You could also make explicit use of setjmp/longjmp. There are many different ways to go here.
- **object orientation, generics, database access, complex numbers, geometric programming, ...** - Really, there is no end of crazy features that you can add to the language.
- **unusual domains** - We’ve been talking about applying LLVM to a domain that many people are interested in: building a compiler for a specific language. However, there are many other domains that can use compiler technology that are not typically considered. For example, LLVM has been used to implement OpenGL graphics acceleration, translate C++ code to ActionScript, and many other cute and clever things. Maybe you will be the first to JIT compile a regular expression interpreter into native code with LLVM?
- **Have fun** - try doing something crazy and unusual. Building a language like everyone else always has, is much less fun than trying something a little crazy or off the wall and seeing how it turns out. If you get stuck or want to talk about it, feel free to email the llvmdev mailing list: it has lots of people who are interested in languages and are often willing to help out.

Before we end this tutorial, I want to talk about some “tips and tricks” for generating LLVM IR. These are some of the more subtle things that may not be obvious, but are very useful if you want to take advantage of LLVM’s capabilities.

8.1 Properties of the LLVM IR

We have a couple common questions about code in the LLVM IR form - lets just get these out of the way right now, shall we?

8.1.1 Target Independence

Kaleidoscope is an example of a “portable language”: any program written in Kaleidoscope will work the same way on any target that it runs on. Many other languages have this property, e.g. lisp, java, haskell, javascript, python, etc (note that while these languages are portable, not all their libraries are).

One nice aspect of LLVM is that it is often capable of preserving target independence in the IR: you can take the LLVM IR for a Kaleidoscope-compiled program and run it on any target that LLVM supports, even emitting C code and compiling that on targets that LLVM doesn’t support natively. You can trivially tell that the Kaleidoscope compiler generates target-independent code because it never queries for any target-specific information when generating code.

The fact that LLVM provides a compact, target-independent, representation for code gets a lot of people excited. Unfortunately, these people are usually thinking about C or a language from the C family when they are asking questions about language portability. I say “unfortunately”, because there is really no way to make (fully general) C code portable, other than shipping the source code around (and of course, C source code is not actually portable in general either - ever port a really old application from 32- to 64-bits?).

The problem with C (again, in its full generality) is that it is heavily laden with target specific assumptions. As one simple example, the preprocessor often destructively removes target-independence from the code when it processes the input text:

```
#ifdef __i386__
  int X = 1;
#else
  int X = 42;
#endif
```

While it is possible to engineer more and more complex solutions to problems like this, it cannot be solved in full generality in a way that is better than shipping the actual source code.

That said, there are interesting subsets of C that can be made portable. If you are willing to fix primitive types to a fixed size (say int = 32-bits, and long = 64-bits), don’t care about ABI compatibility with existing binaries, and are willing to give up some other minor features, you can have portable code. This can make sense for specialized domains such as an in-kernel language.

8.1.2 Safety Guarantees

Many of the languages above are also “safe” languages: it is impossible for a program written in Java to corrupt its address space and crash the process (assuming the JVM has no bugs). Safety is an interesting property that requires a combination of language design, runtime support, and often operating system support.

It is certainly possible to implement a safe language in LLVM, but LLVM IR does not itself guarantee safety. The LLVM IR allows unsafe pointer casts, use after free bugs, buffer over-runs, and a variety of other problems. Safety needs to be implemented as a layer on top of LLVM and, conveniently, several groups have investigated this. Ask on the llvmdev mailing list if you are interested in more details.

8.1.3 Language-Specific Optimizations

One thing about LLVM that turns off many people is that it does not solve all the world's problems in one system (sorry 'world hunger', someone else will have to solve you some other day). One specific complaint is that people perceive LLVM as being incapable of performing high-level language-specific optimization: LLVM "loses too much information".

Unfortunately, this is really not the place to give you a full and unified version of "Chris Lattner's theory of compiler design". Instead, I'll make a few observations:

First, you're right that LLVM does lose information. For example, as of this writing, there is no way to distinguish in the LLVM IR whether an SSA-value came from a C `int` or a C `long` on an ILP32 machine (other than debug info). Both get compiled down to an `i32` value and the information about what it came from is lost. The more general issue here, is that the LLVM type system uses "structural equivalence" instead of "name equivalence". Another place this surprises people is if you have two types in a high-level language that have the same structure (e.g. two different structs that have a single int field): these types will compile down into a single LLVM type and it will be impossible to tell what it came from.

Second, while LLVM does lose information, LLVM is not a fixed target: we continue to enhance and improve it in many different ways. In addition to adding new features (LLVM did not always support exceptions or debug info), we also extend the IR to capture important information for optimization (e.g. whether an argument is sign or zero extended, information about pointers aliasing, etc). Many of the enhancements are user-driven: people want LLVM to include some specific feature, so they go ahead and extend it.

Third, it is possible and easy to add language-specific optimizations, and you have a number of choices in how to do it. As one trivial example, it is easy to add language-specific optimization passes that "know" things about code compiled for a language. In the case of the C family, there is an optimization pass that "knows" about the standard C library functions. If you call "exit(0)" in main(), it knows that it is safe to optimize that into "return 0;" because C specifies what the 'exit' function does.

In addition to simple library knowledge, it is possible to embed a variety of other language-specific information into the LLVM IR. If you have a specific need and run into a wall, please bring the topic up on the llvmdev list. At the very worst, you can always treat LLVM as if it were a "dumb code generator" and implement the high-level optimizations you desire in your front-end, on the language-specific AST.

8.2 Tips and Tricks

There is a variety of useful tips and tricks that you come to know after working on/with LLVM that aren't obvious at first glance. Instead of letting everyone rediscover them, this section talks about some of these issues.

8.2.1 Implementing portable offsetof/sizeof

One interesting thing that comes up, if you are trying to keep the code generated by your compiler "target independent", is that you often need to know the size of some LLVM type or the offset of some field in an llvm structure. For example, you might need to pass the size of a type into a function that allocates memory.

Unfortunately, this can vary widely across targets: for example the width of a pointer is trivially target-specific. However, there is a clever way to use the getelementptr instruction that allows you to compute this in a portable way.

8.2.2 Garbage Collected Stack Frames

Some languages want to explicitly manage their stack frames, often so that they are garbage collected or to allow easy implementation of closures. There are often better ways to implement these features than explicit stack frames, but LLVM does support them, if you want. It requires your front-end to convert the code into Continuation Passing Style and the use of tail calls (which LLVM also supports).